NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

IMPROVING SYNTACTIC MATCHING FOR MULTI-LEVEL FILTERING

by

Jeffrey S. Herman

September, 1997

Thesis Advisor:	V. Berzins
Co-Advisor:	Luqi

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1997	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE: IMPROVING SYNTACTIC MATCHING FOR MULTI-LEVEL FILTERING			5. FUNDING NUMBERS
6. AUTHOR(S) Herman, Jeffrey S.			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

At the center of software reuse is the search and retrieval of software components from large software libraries. Recent research has illuminated a promising approach called *multi-level filtering* that breaks the problem up into a series of increasingly stringent filters that move along a continuum of high-recall, low-precision syntactic techniques towards the more computationally expensive, high-precision semantic techniques.

In multi-level filtering, syntactic matching is decomposed into two phases: profile filtering and signature matching. This thesis presents improvements to the resolution of syntactic profiles where the intent is to increase precision without a loss in recall during profile filtering. Large integer representation of profiles and profile lookup tables lead to an optimal time-and-space solution to profile representation. Finally, a new approach to signature matching is proposed that provides early pruning of the search-space in an effort to cut down the time it takes to find valid signature maps.

The resulting software is mature enough for future integration with the other elements of multi-level filtering as well as inclusion in a CASE tool such as CAPS.

14. SUBJECT TERMS Software Reuse, Syntactic Matching, Signature Matching, Multi-level Filtering			15. NUMBER OF PAGES 146
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

IMPROVING SYNTACTIC MATCHING FOR MULTI-LEVEL FILTERING

Jeffrey S. Herman
B.A., Computer Science, University of California San Diego, 1990

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1997

Author: [illegible]
Jeffrey S. Herman

Approved by: Valdis Berzins
V. Berzins, Thesis Advisor

[illegible]
Luqi, Co-Advisor

[illegible]
T. Lewis, Chairman
Department of Computer Science

ABSTRACT

At the center of software reuse is the search and retrieval of software components from large software libraries. Recent research has illuminated a promising approach called *multi-level filtering* that breaks the problem up into a series of increasingly stringent filters that move along a continuum of high-recall, low-precision syntactic techniques towards the more computationally expensive, high-precision semantic techniques.

In multi-level filtering, syntactic matching is decomposed into two phases: profile filtering and signature matching. This thesis presents improvements to the resolution of syntactic profiles where the intent is to increase precision without a loss in recall during profile filtering. Large integer representation of profiles and profile lookup tables lead to an optimal time-and-space solution to profile representation. Finally, a new approach to signature matching is proposed that provides early pruning of the search-space in an effort to cut down the time it takes to find valid signature maps.

The resulting software is mature enough for future integration with the other elements of multi-level filtering as well as inclusion in a CASE tool such as CAPS.

TABLE OF CONTENTS

I. INTRODUCTION

Effective software reuse becomes increasingly more important as the cost and complexity of software development escalates. At the center of the issue is the search and retrieval of software components from large software libraries. When enterprises that encourage the creation of reusable software components succeed in their efforts they are often met with the discouraging reality that large software bases are difficult to *use*. Issues such as query formulation, component storage, component retrieval, and presentation of query results must all be addressed with the same technology/usability tradeoffs that accompany most tools. Searching and retrieving components in large software bases has typically been plagued by poor recall and precision, slow algorithms, and demanding query requirements.

In an effort to address such shortcomings, the literature has shed light on numerous techniques for searching and retrieving components in large software bases but they usually fall short due to their narrow approaches. A promising new hybrid approach called *multi-level filtering* combines many of the traditional aspects of search and retrieval, such as keyword matching, syntactic matching, and semantic matching. The method breaks the problem up into a series of increasingly stringent filters that move along a continuum of high-recall, low-precision syntactic techniques towards the more computationally expensive, high-precision semantic techniques. This thesis is focused on improving the syntactic matching filters used early in the process of multi-level filtering.

To begin, section II reviews the relevant literature leading up to multi-level filtering. Section III more specifically discusses the architecture of multi-level filtering, including the decomposition of syntactic matching into its two phases of profile filtering and signature matching. Section IV presents improvements to the resolution of syntactic profiles[1] that can increase precision without a loss in recall during profile filtering. Also presented are improvements to the internal representation of syntactic profiles that lead to an optimal time-and-space solution to profile representation. Section V outlines improvements to signature matching that provide early pruning of the search-space in an

[1] Unique to the multi-level filtering method, a *syntactic profile* is a normalized representation of a software component's syntactic properties. A more detailed definition is found in section IV.

effort to cut down the time it takes to find valid signature maps. Section VI discusses the effectiveness of the improvements through a series of experiments. Section VII draws some conclusions and suggests areas for future research. The last sub-section in sections IV and V contain a detailed design of the improvements and the appendix contains the source code representing the design's implementation. The resulting software is mature enough for future integration with the other elements of multi-level filtering as well as inclusion in a CASE tool such as CAPS.

II. BACKGROUND

A sampling of previous work in software component search and retrieval is presented in this section to provide some background and basis for the ideas proposed in this thesis.

A. KEYWORD MATCHING

The classical and somewhat popular approach to software search and retrieval has been the employment of keyword matching. Components are assigned keywords that describe their attributes and functionality. Queries are specified with keywords and a simple search through the software base for components with matching keywords returns the candidate set of components. Such an approach breaks down, however, as the size of the software base increases. A large set of keywords can cause loss of recall and small sets of keywords can cause loss of precision.

[11] improves on the classical keyword technique by utilizing a faceted approach that better structures the terms used for classifying the components. Terms chosen from a set of facets are used to categorize all the components. This facilitates a closer fit of terms and reduces the problem of deciding the best keyword to use from a fixed set of standard keywords.

Among the problems with keyword-based approaches is the inherent requirement of a well-versed librarian. The infamous garbage-in/garbage-out principle certainly applies to the software base population activity. If the librarian does not have appropriate domain knowledge for each component admitted into the software base then the keywords will not be chosen correctly and penalties in recall and precision during search and retrieval will ensue.

A long overdue use of keyword matching is to apply it along side other techniques. The multi-level filtering method in [9] is an example of such a hybrid approach. The results of keyword matching are summarized in a computed keyword ratio that can be used to determine if a candidate should be forwarded to the next filter. If problems with recall and precision emerge, the keyword filter threshold can be adjusted or the keyword filter can be deactivated altogether.

B. SYNTACTIC MATCHING

Syntactic matching has been proposed as an effective method for quickly ruling out components that cannot match the query [13]. The process can be successfully automated when syntactic normalizing procedures are applied. Syntactic normalization procedures come in many forms [2][6][13] but perhaps the most promising approach proposed recently is the application of syntactic profiles [9]. This approach is discussed and improved upon in section IV of this thesis.

The presence of subtypes in queries and components has often plagued syntactic matching by imposing penalties in recall. For example, if the query is an operation that takes a *positive* as an input and the operation components in the software base only contain *integer* inputs then the query will fail even though positives are legitimate subtypes of integers. Such a shortcoming is addressed in [2] and further refined in [9].

C. SEMANTIC MATCHING

A major shortcoming of syntactic matching is its inability to retrieve components based on their behavior. If syntactic matching were the sole approach to search and retrieval a query for a square-root function would indeed return a square-root function but, to the user's dismay, most of the other math functions in the software base would be returned as well! Recent efforts have attempted to address this shortcoming through various approaches to semantic matching. Specification-based approaches found in [6] and [8] require the user to form queries as behavioral specifications but haven't been met with great success due to the difficulty of forming correct specifications.

The approach of using algebraic specifications [13] for encoding a component's behavior has led to promising results for successfully automating semantic matching [9]. A set of ground equations describing the component's behavior can be specified algebraically using algebraic specification languages such as OBJ3 [3] and included with the component. The terms in the equations can be applied from left to right to simplify them to their canonical form where they can then be easily compared to a query's set of ground equations. Algebraic specifications, however, are not much of an improvement with regards to ease of use. Specification languages such as OBJ3 have to be absorbed by the librarian and the user and the domain of the component needs to be understood. A librarian will be met with a cumbersome task when preparing an entire software base for this type of semantic matching [10].

D. MULTI-LEVEL FILTERING

Multi-level filtering [9] is an approach that integrates keyword, syntactic, and semantic matching. It is attractive because it applies a series of increasingly stringent filters that move along a continuum of high-recall, low-precision syntactic techniques towards the more computationally expensive, high-precision semantic techniques. The purpose of the work described in this thesis is to improve upon the syntactic matching processes of multi-level filtering. Hence, a discussion regarding the specifics of the multi-level filtering approach will be postponed to their relevant sections of this document.

E. SOFTWARE BASE DESIGN AND POPULATION

Populating the software base usually involves annotating the components with additional information to facilitate search and retrieval. In every approach cited above this is the case. PSDL [5] has been shown to be an effective language for representing components independently of their native language [10]. In addition to its real-time specification support, PSDL supports operations (including generic operations), abstract data types (including generic types), state machines, and the common predefined types found in most popular programming languages. Thus PSDL is more than sufficient for representing the syntactical properties of queries and reusable components in a software base. PSDL also provides a placeholder for axioms to provide semantic information for the component. Algebraically specified ground equations in the form of OBJ3, for instance, can be placed in this section of the PSDL file.

CAPS [7], a CASE tool for rapid prototyping of embedded hard real-time systems, represents great strides in integrating modern software engineering technologies. The system includes a graphical editor, an execution support system, an evolution control system, automated real-time schedulers, automated integration of Ada modules, and placeholders for making use of a software base. Its initial software base [10] includes reusable components from the Booch library. The components include syntactic specifications in PSDL and semantic specifications in OBJ3 thereby providing a good test suite for multi-level filtering and the ideas proposed in this thesis.

III. MULTI-LEVEL FILTERING ARCHITECTURE

The model of multi-level filtering is illustrated in Figure 1. The entire process can be generalized into two main activities: syntactic matching and semantic matching. Syntactic matching quickly filters out candidates based on syntactic properties to eliminate as many candidates as possible that must undergo the computationally expensive semantic matching. Clearly it is advantageous to filter out large numbers of candidates early to minimize the use of the more laborious filters later in the process. At any stage of the process the user should be able to set the thresholds that determine the constraints within which a candidate may pass. Furthermore, the user should be able to browse the set of candidates from the prior filters and have the option of manually filtering the results that are passed to the next filter.

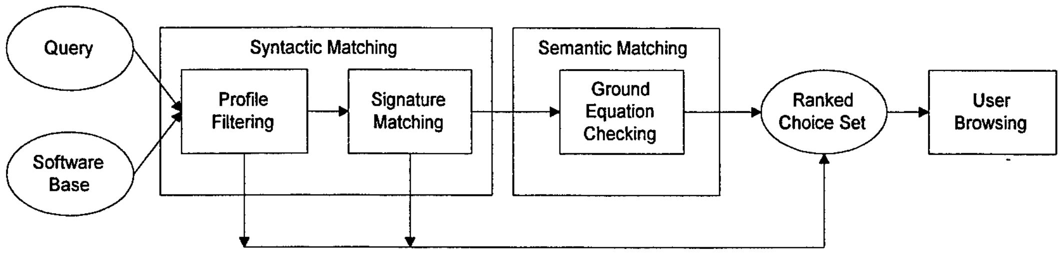


Figure 1: Multi-level Filtering Model

This thesis focuses on improving syntactic matching by making improvements to profile filtering and signature matching. Section IV discusses improvements to profile filtering and section V discusses improvements to signature matching. In addition to the presentation of theoretical improvements, each section also covers a detailed design and implementation for realizing a software module that can be practically used within the entire context of multi-level filtering and ultimately in CAPS.

IV. PROFILE FILTERING

A. CURRENT STATE OF THE ART

In [9] a component's syntactic properties are represented as a *Component Profile.* A component profile is the multiset of *Operation Profiles* for all the operations in a component. An operation profile is a sequence of integers each representing a unique syntactic property[2] of an operation. Definition 4 in [9] defines an operation profile as:

1. The first integer is the total number of occurrences of sorts.
2. If the total number of sort groups, N, is greater than 0, then the second to (1 + N)th integers are the cardinalities of the sort groups, in descending order.
3. The (2+N)th integer is the cardinality of the unrelated sort group.
4. The (3+N)th integer is:

   0 if the value sort is different from any of the argument sorts; and

   1 if the value sort belongs to some sort group.

By computing the component profiles for each reusable component in the software base, components can be placed into partitions where each partition is identified by the component profile of the components it contains. An ordering of these partitions can then be obtained to organize the software base into a haase-diagram for facilitated traversal during a process [9] defines as *Profile Filtering.*

Profile filtering is a process in which components in the software base can be easily ruled out based on whether their syntactic profiles match the query's syntactic profile. This is a high-speed (relative to signature and semantic matching) process where the goal is to increase precision in a typically high-recall/low-precision stage of retrieval.

[2] These properties have been referred to as profile *components* but we will use the term *property* rather than *component* to eliminate an overloading of the term *component* which we have been using to refer to a reusable component such as a type.

B. PROFILE IMPROVEMENTS

One way to increase precision in [9]'s approach to profile filtering is to make improvements to the definition of an operation profile. Two categories of improvement that can be easily quantified are *Resolution* and *Space-and-Time.*

1. Resolution

The point of increasing the resolution of syntactic profiles is to better distinguish between syntactically similar software components. In terms of [9]'s architecture this would result in an increase in the number of partitions in the software base. In terms of [9]'s profile filtering process this would mean an increase in the number of nodes in the haase-diagram that maps the software base's organization.

Gains in resolution can be obtained two ways:

1. Add more properties to the profile.
2. Use properties that can be measured with more possible values.

In keeping with the spirit of syntactic normalization, however, one has to be careful to define measurements that will not be affected by the permutation of the arguments or by any renaming of the types.

[1] inspired several resolution improvements to profiles that can prove quite useful in partitioning the software base more effectively. The first improvement follows the second resolution-gain technique described above and the other improvements subscribe to the first technique.

a. Value Sort Frequency

Item 4 of [9]'s operation profile definition has two possible values, 0 or 1, indicating if the value sort[3] is in the same sort group as other arguments in the operation or if it is a member of the unrelated sort group. The resolution of this particular property can be increased by modifying its definition to be the number of occurrences of the value sort in the operation's signature. Table 1 illustrates this improvement:

Table 1

Component Operations	Old	New
add: id set → set	1	2
union: set set → set	1	3
member: id set → bool	0	1
choose: set → id	0	1

The increase in resolution is well illustrated with the operations *add* and *union*. When using the old definition we notice that *add* and *union* have the same value for the value-sort property. When using the new definition we see that *add* and *union* each have different measurements for this property. Such a difference guarantees that *add* and *union* will have different profiles and therefore contributes to an increased resolution of the software base.

b. Type Sort Frequency

A majority of the reusable components the author has come across have been abstract data types. In most cases these types refer to themselves in the operations they define. For instance, in Table 1 the component is a *set* and one will notice the operations refer to *set* frequently. The frequency of such self-references can be measured and can contribute to the component's profile. Table 2 illustrates the additional property:

[3] The term *value sort* is used by [9] to refer to the type of the output argument of an operation. In this thesis the terms *value* and *output* are used interchangeably but an effort to use *value* when referring to concepts in [9] will be made.

Table 2

Component Operations	Old	New
add: id bag $\rightarrow$ bag	n/a	2
merge: bag bag $\rightarrow$ bag	n/a	3
equal: bag bag $\rightarrow$ bool	n/a	2
equal_with_set: bag set $\rightarrow$ bool	n/a	1
member: id bag $\rightarrow$ bool	n/a	1
freq: id bag $\rightarrow$ natural	n/a	1

The new property measures the number of times *bag* is referred to in the operation's signature. Notice that equal and equal_with_set are assigned different values for this new property. In the old profile definition, these two operations would have the same profile. Again, we have an improvement in resolution.

c. Predefined Sort Frequencies

The final resolution improvement to introduce involves representing the sizes of the various sort groups for the predefined[4] types. [9] and [2] both note that during signature matching the predefined types can only map to predefined types of the same sort group.[5] Given this requirement, it would be beneficial to filter out components during profile filtering that would violate such a requirement. Hence we can add an integer for each predefined sort group that would reflect the size of that sort group in the operation's signature. In Table 3, five predefined sort groups are recognized in the following order: boolean, character, string, integer, and real.

Table 3

Component Operations	Old	New
add: id bag $\rightarrow$ bag	n/a	00000
merge: bag bag $\rightarrow$ bag	n/a	00000
member: id bag $\rightarrow$ bool	n/a	10000
freq: id bag $\rightarrow$ natural	n/a	00010

The operations *member* and *freq* are good examples of the increased resolution this improvement provides. The old profile definition would assign these two

[4] [9] refers to predefined types as *basic* types. The two terms are used interchangeably in this thesis.
[5] [2] further restricts this statement with rules regarding subtype matching within the sort group. This is addressed in the section on Signature Matching where the discussion is more applicable.

operations the same profile. The enhancement assigns different profiles thereby increasing the resolution and eliminating the signature-matching algorithm from trying to map incompatible predefined types.

2. Time-and-Space

Software bases can become enormous rather quickly. A large enterprise's software base can contain thousands of reusable components. Representing such a large software base in the architecture proposed by [9] can tax the resources of the enterprise's computer/s responsible for maintaining and searching the software base. To this end, the representation of syntactic profiles is an issue worth special attention since thousands of components can actually translate into tens of thousands of operations!

[9] suggests the operation profile be represented as a sequence of integers. This requires a sequence abstract data type with standard operations defined such as equality and less-than (for sorting). Numerous instantiations of such an abstract data type could require a substantial amount of memory. Two possible suggestions for making time-and-space improvements to syntactic profile representation are explained below.

a. Large Integer Representation

A representation that would take up less space would be a large integer of something like 64 bits. Each digit in the integer would represent each integer in the profile. Besides space, speed issues regarding the testing for equality and less-than would be greatly sped up because the default operations for the integer would apply, thereby eliminating the need for putting a user-defined function on the stack each time these common operations are called.

The biggest disadvantage to this approach should be evident: such a representation would limit the number of sort occurrences in the signature to nine. A function with ten sort occurrences is rather rare, however. One could use two digits for each property thereby potentially relaxing the restriction to 99 sort occurrences, which is definitely enough. Two digits per property, however, would require a much larger integer than one that could be represented with 64 bits and it is questionable if any high-level language can efficiently represent greater-than-64-bit integers any more efficiently than a smart implementation of a sequence.

b. Profile Lookup Table

A component profile is traditionally thought of as a sequence of operation profiles. In other words, it is a sequence of sequences of integers. Given thousands of components, this can take up a lot of space and can tax the component profile equality operations. Especially wasteful is the fact that the number of unique operation profiles is much smaller than the actual number of components in the software base.

A promising approach for improving the time-and-space issues of component profiles is the employment of a profile lookup table. To eliminate the redundancy of integer sequences that represent operation profiles throughout the software base, this table would map a unique integer to each unique operation profile used in the software base. A component profile can then be represented as sequence of these unique integers rather than a sequence of integer sequences. Below is an example to illustrate the concept:

Table 4: Profile Lookup Table

Lookup ID	Operation Profile
1	[2,1,2,1,0,0,0,0,0]
2	[3,1,3,1,0,0,0,1,0]
3	[3,1,3,1,1,0,0,0,0]
4	[3,2,1,2,0,0,0,0,0,2]
5	[3,3,0,3,0,0,0,0,0,3]

Table 5: Set

Component Operations	Operation Profiles	Lookup ID
add: id set → set	[3,2,1,2,0,0,0,0,0,2]	4
union: set set → set	[3,3,0,3,0,0,0,0,0,3]	5
member: id set → bool	[3,1,3,1,1,0,0,0,0]	3
choose: set → id	[2,1,2,1,0,0,0,0,0]	1

Table 6: Bag

Component Operations	Operation Profiles	Lookup ID
add: id bag → bag	[3,2,1,2,0,0,0,0,0,2]	4
merge: bag bag → bag	[3,3,0,3,0,0,0,0,0,3]	5
member: id bag → bool	[3,1,3,1,1,0,0,0,0]	3
freq: id bag → natural	[3,1,3,1,0,0,0,1,0]	2

Table 4 depicts the profile lookup table after the *set* and *bag* components from Table 5 and Table 6 have been loaded. The first thing to note from this example is the redundancy in operation profiles between *set* and *bag*. Three out of the five unique operation profiles in the lookup table are shared between *set* and *bag*. The second thing to note is the huge space savings gained for a component profile. Without the lookup table the *set's* component profile would be [[3,2,1,2,0,0,0,0,0,2], [3,3,0,3,0,0,0,0,0,3], [3,1,3,1,1,0,0,0,0], [2,1,2,1,0,0,0,0,0]]. By using the lookup table *set's* component profile can be represented as [4,5,3,1].[6] Given thousands of components the amount of space saved is significant. Furthermore, the amount of time saved checking for component profile equality can be substantial since the number of actual integer comparisons is cut drastically.

The profile lookup table represents an optimal time-and-space solution to profile filtering. During profile filtering, the actual profiles themselves are irrelevant. What is relevant is whether two profiles are the same. The profile lookup table ensures that each profile is represented by a unique identifier. Since this identifier can be represented by an integer we have an optimal time-and-space solution to profile representation.

C. DESIGN AND IMPLEMENTATION

The software used in [9] is not very conducive to reusability and extendibility and therefore is difficult to use for testing the improvements in syntactic matching outlined in this thesis. Furthermore, it is desirable to have a software module that is practical for inclusion in CAPS. To this end, a significant amount of design and implementation is necessary. This section details the various data types and implementation strategies used to implement a practical system to test the improvements proposed in this thesis with the understanding that such a system should ultimately integrate with other elements of multi-level filtering and CAPS in the large.

1. Component Types

[13] proposed components ultimately be stored in an object-oriented database to easily associate the various elements of a software component required for search and retrieval. This idea is highly appropriate for a production quality implementation of a software base but given the lack of engineering resources at this stage of the research such an idea has not yet come to fruition. The CAPS software base is currently composed of a set of files for each component where each file for the component represents a different element of the component that is useful for reuse [10]. Specifically this includes the component's native language (e.g. Ada) specification, native language body, PSDL specification, and OBJ3 specification.

The first task, then, is to organize these files into an intelligent scheme to support the goals of this thesis and the short-term goals of the CAPS project. The organization proposed here is to create a directory for each component that contains all of its files. Figure 2 illustrates examples of these directories.

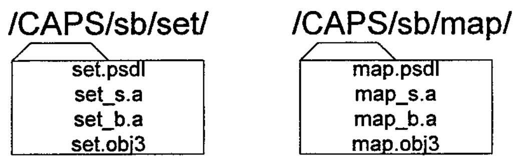
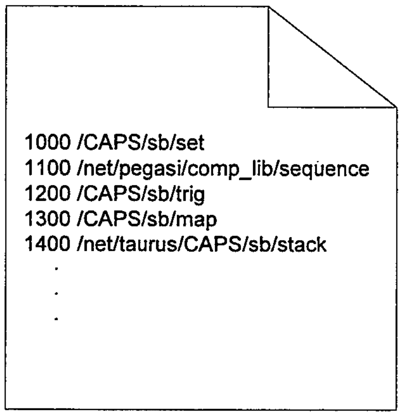


Figure 2: Sample Directories for a Software Base

A header file is used to identify all of the components that comprise the software base. An example of such a header file is shown in Figure 3. Notice that a unique integer is

[6] The component profile in this example is not ordered but could be for improved signature matching. A discussion of this can be found in section V.B.1.

assigned to each component. This ID will be used to identify the component in the data structures that internally represent the software base because it is easier to manipulate and it saves space. A nice feature the header file provides is the ability to represent a distributed software base due to the use of a networked file system. Notice components 1100 and 1400 are components that actually exist on remote machines.



```
1000 /CAPS/sb/set
1100 /net/pegasi/comp_lib/sequence
1200 /CAPS/sb/trig
1300 /CAPS/sb/map
1400 /net/taurus/CAPS/sb/stack
   .
   .
   .
```

Figure 3: Sample Header File for a Software Base

Now that we have a way of representing the components in secondary storage we need a way of representing them internally. Figure 4 shows the objects used to represent components in memory using Rational's Unified Method [12].

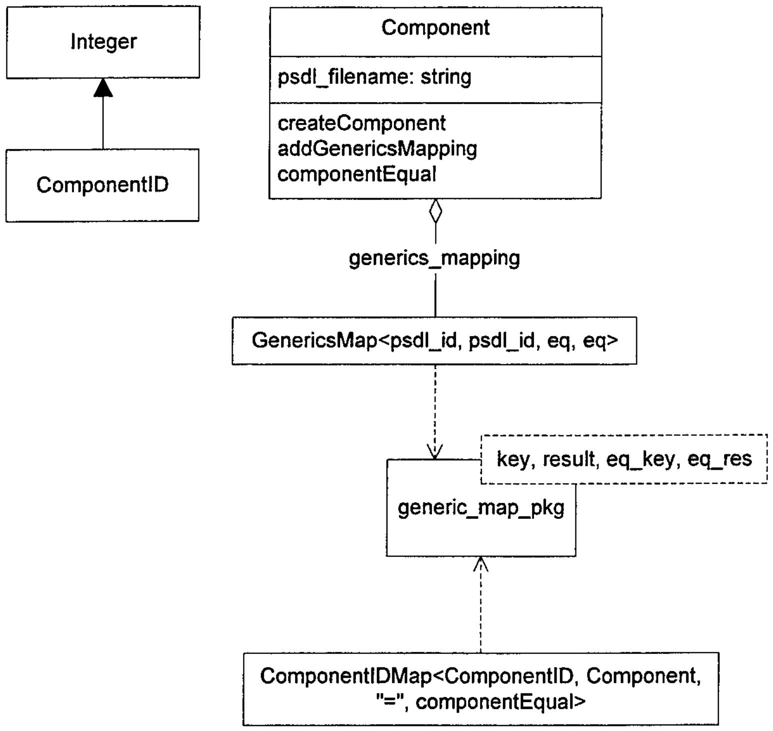


Figure 4: Component Types

The ComponentIDMap maps the ComponentID to a Component. The ComponentID is the unique integer read in from the software base's header file. The Component contains the filename of the component's PSDL specification and an association to an instance of a GenericsMap. A GenericsMap maps the generic parameter identifiers in generic components to actual type names. This needs some explanation.

Suppose we have the following PSDL specification for the generic component Stack:

```
TYPE Stack
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE

OPERATOR Push
  SPECIFICATION
    INPUT
      The_Item : Item,
      On_The_Stack : Stack
    OUTPUT
      On_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END
```

```
OPERATOR Pop
SPECIFICATION
  INPUT
    The_Stack : Stack
  OUTPUT
    The_Stack : Stack
  EXCEPTIONS
    Overflow, Underflow
END
```

```
OPERATOR Depth_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow
  END
```

```
  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

END
```

This component has one generic parameter named *Item* and makes reference to three different types: *Stack*, *Natural*, and *Boolean*. Instantiating Item to the different types used in the component can potentially yield a different component profile for each instantiation. This could place the various instantiations into different partitions. Hence, each generic component must undergo the generic instantiation process to obtain the various generic parameter mappings. Each instantiation is stored internally as a separate component with its unique generic mapping. The ComponentID for each instantiation is based on the base ID from the header file. For example, if the header file assigns the ID 1200 for the stack component listed above then the ComponentID entries in the ComponentIDMap would be 1201, 1202, 1203, and 1204. Table 7 illustrates this mapping.

Table 7

ComponentID	Component	Component.generic_mapping
1201	Stack	Item → Stack
1202	Stack	Item → Natural
1203	Stack	Item → Boolean
1204	Stack	Item → Item

Notice there is a fourth entry for mapping Item to itself. This is a simple way of representing the possibility that the generic parameter does not map to any of the types used in the component. Another important point to note is the ids in the header file need to be spaced sufficiently to give the generic instantiation algorithm room for the automatic generation of unique ids for a given component. The software base used to test the ideas in this thesis was given a spacing of 100 between component ids, which provided sufficient room for generic instantiation.

One final point to note regarding generic parameter instantiation is a single generic component can end up being instantiated into numerous components through the generic parameter instantiation process. This is especially true for components with more than one generic parameter because the cross-product of the generic parameters and the normal types in the component must be computed to exhaust all the possible combinations. Measurements regarding the instantiation of generic components are presented in section VI.A.

2. Profile Types

The natural design for profiles and component profiles is to use sequences. A Profile would be implemented as a sequence of integers and a ComponentProfile would be implemented as a sequence of Profiles. This approach is depicted in Figure 5.

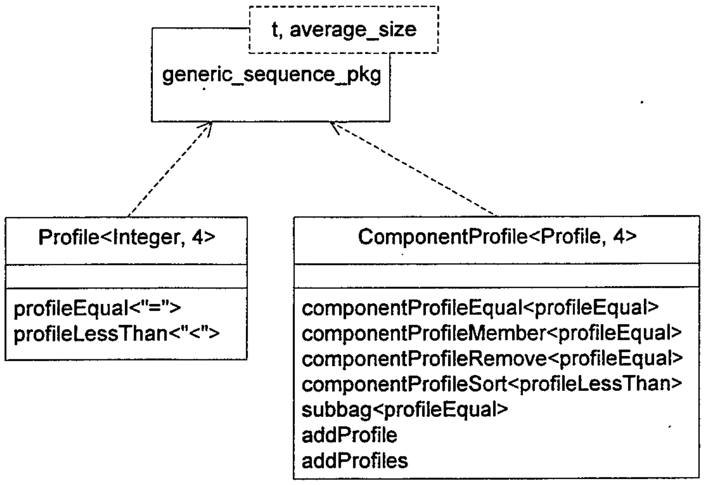
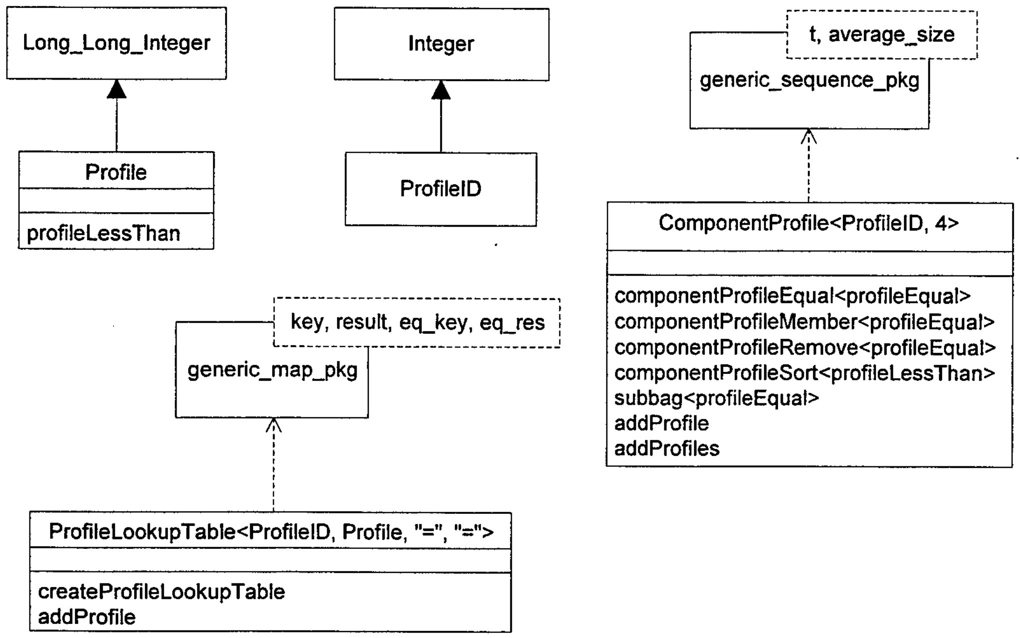

t, average_size
generic_sequence_pkg
Profile<Integer, 4>
profileEqual<"=">
profileLessThan<"<">
ComponentProfile<Profile, 4>
componentProfileEqual<profileEqual>
componentProfileMember<profileEqual>
componentProfileRemove<profileEqual>
componentProfileSort<profileLessThan>
subbag<profileEqual>
addProfile
addProfiles

Figure 5: Profile Data Types

The method *profileLessThan* provides a means of ordering the Profiles lexicographically in the ComponentProfile. The advantages of such an ordering are detailed in section V.B.1. The method *subbag* is a multiset subset operation that can be used to order the partitions in the haase diagram since the partitions are keyed using ComponentProfiles. The design and implementation of the haase diagram is discussed in section IV.C.3.

Section IV.B.2 introduced time-and-space improvements to the design in Figure 5. These improvements are represented in Figure 6.



Long_Long_Integer
Integer
t, average_size
generic_sequence_pkg
Profile
profileLessThan
ProfileID
ComponentProfile<ProfileID, 4>
componentProfileEqual<profileEqual>
componentProfileMember<profileEqual>
componentProfileRemove<profileEqual>
componentProfileSort<profileLessThan>
subbag<profileEqual>
addProfile
addProfiles
key, result, eq_key, eq_res
generic_map_pkg
ProfileLookupTable<ProfileID, Profile, "=", "=">
createProfileLookupTable
addProfile

Figure 6: Time-and-Space Improvements for Profile Types

This shows all of the improvements used together but it is possible through the use of abstract data types to mix and match the designs. For example, if one wanted to be able to handle operations with more than nine arguments (see section IV.B.2.a) then Profile could be implemented as a sequence of integers rather than the Long_Long_Integer and still be able to take advantage of the ProfileLookupTable.

3. Haase Diagram Types

The haase diagram can be constructed using the objects in Figure 7.

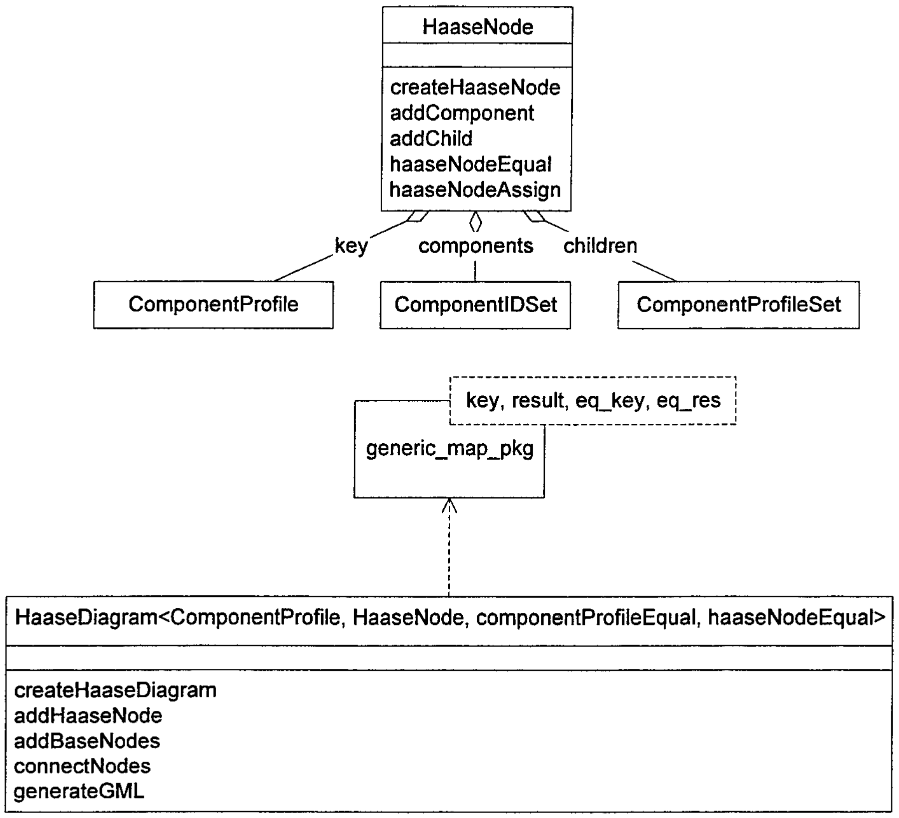


Figure 7: Haase Diagram Data Types

A HaaseNode is a partition that is keyed by a ComponentProfile. The node contains components that have the same ComponentProfile as the key. Notice the components are a set of ComponentIDs rather than Components to save space. When access to the actual component is necessary the ComponentID can be used to fetch the component from the ComponentIDMap as described in section IV.C.1. The HaaseNode is related to other nodes (or partitions) through its *children* association. This association is implemented as a set of ComponentProfiles, which are the *keys* to the next partitions in the ordering. Relating nodes in this way allows the use of a map to represent the entire haase diagram. Direct access to partitions can be obtained by fetching with a ComponentProfile key.

Constructing the haase diagram is a three step process.

Step 1: for each component check if a node exists with that component's CompoentProfile. If it does then put that component in that node (add it to the node's *components* association). Otherwise add a new node with the component's ComponentProfile as the key and put the component in it.

Step 2: for each Profile in each HaaseNode's key add a node to represent a base node. This is accomplished by calling *addBaseNodes* on the populated HaaseDiagram from step 1.

Step 3: connect the nodes (set the *children* association for each node) based on the following invariant from [9]: n2 is n1's child if and only if *subbag(n1.key, n2.key)* and there is no node n3 such that *subbag(n1.key, n3.key)* and *subbag(n3.key, n2.key)*. This is accomplished by calling *connectNodes* on the populated HaaseDiagram from step 2.

4. Candidate Types

Candidates are the "currency" passed between the various stages of the multi-level filtering process. Figure 8 shows these data types.

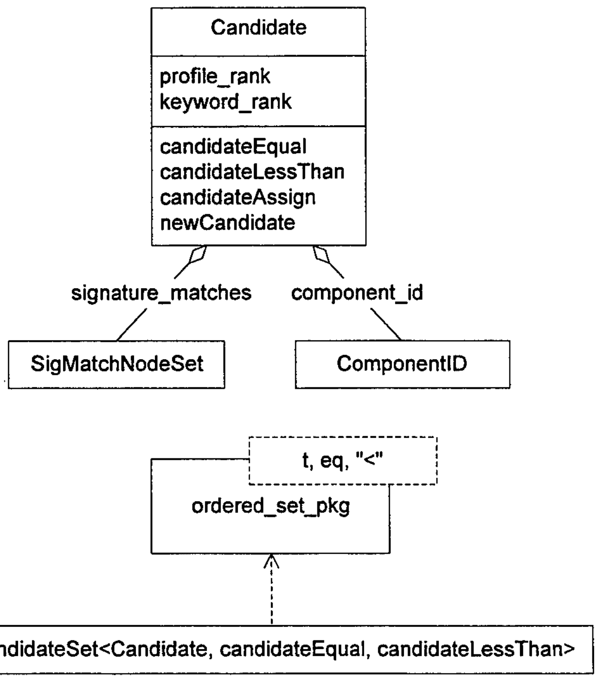


Figure 8: Candidate Data Types

A Candidate is a component with a ranking. The component is identified through the ComponentID association. The ranking is a combination of the results of profile filtering, keyword filtering, and signature matching. [9] calls this combination the *KPS* value. Each candidate can have multiple signature matches, each with a different signature rank, so an association to a SigMatchNodeSet is present (see section V.C). Notice the CandidateSet is an *ordered* set. The ordering is provided through the candidateLessThan method which uses the KPS value to determine the ordering.

5. Software Base Types

The software base ties everything together. The *software_base* object and the functional summary of its methods are shown in Figure 9. The *initialize* method is responsible for parsing the header file, loading the components' PSDL specifications, generating the generics mappings for the generic components, computing the profiles, and populating the haase diagram and component id map. The software base also provides some methods for gathering statistics, including a method to generate a GML [4] file to graphically depict the haase diagram. Finally, the software base contains methods for profile filtering and signature matching.

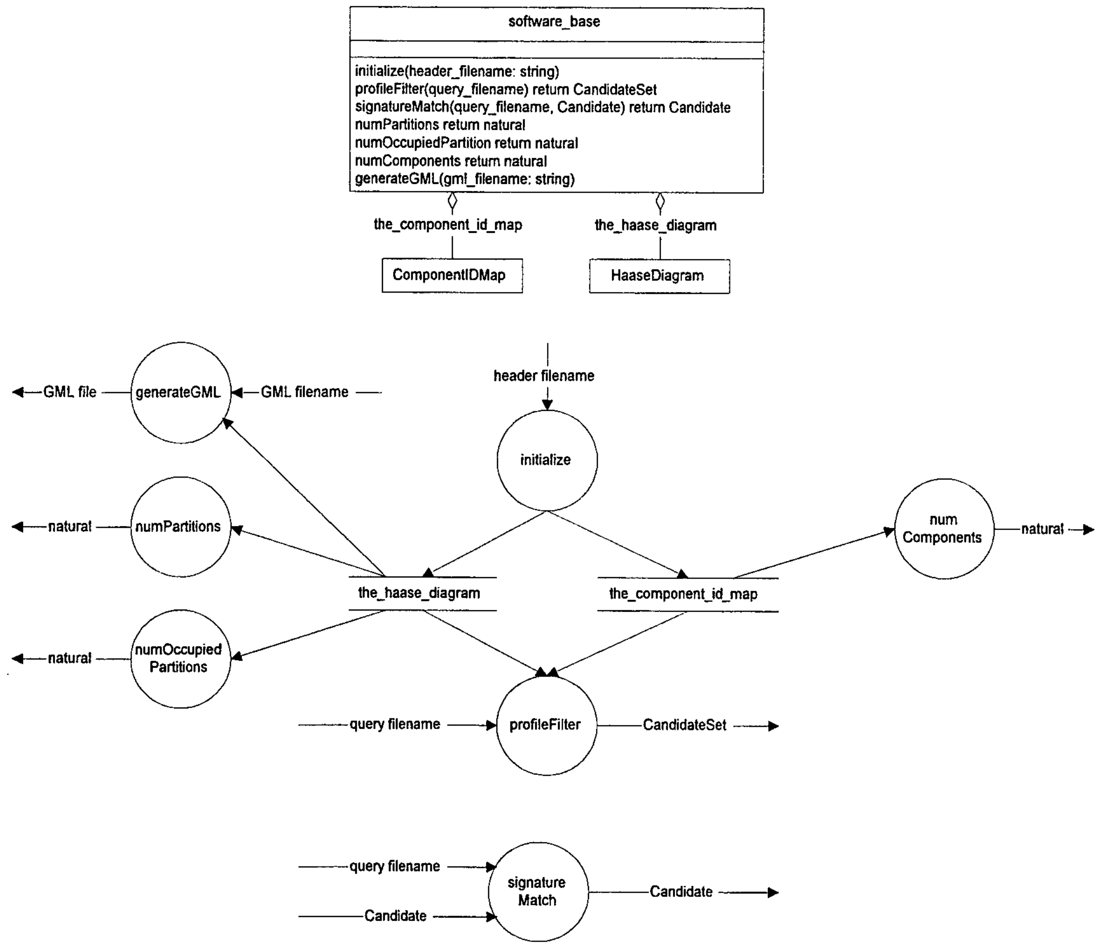


Figure 9: Software Base Types and Functions

6. Profile Filtering Strategy

We now have an infrastructure with which to experiment and conduct profile filtering and have laid the groundwork for a signature matching implementation which is presented in section V.C. The profile filtering strategy laid out in [9] can now be applied to this design and is encapsulated in the method *profileFilter*. A high level expansion of the profileFilter method in the software base is shown in Figure 10.

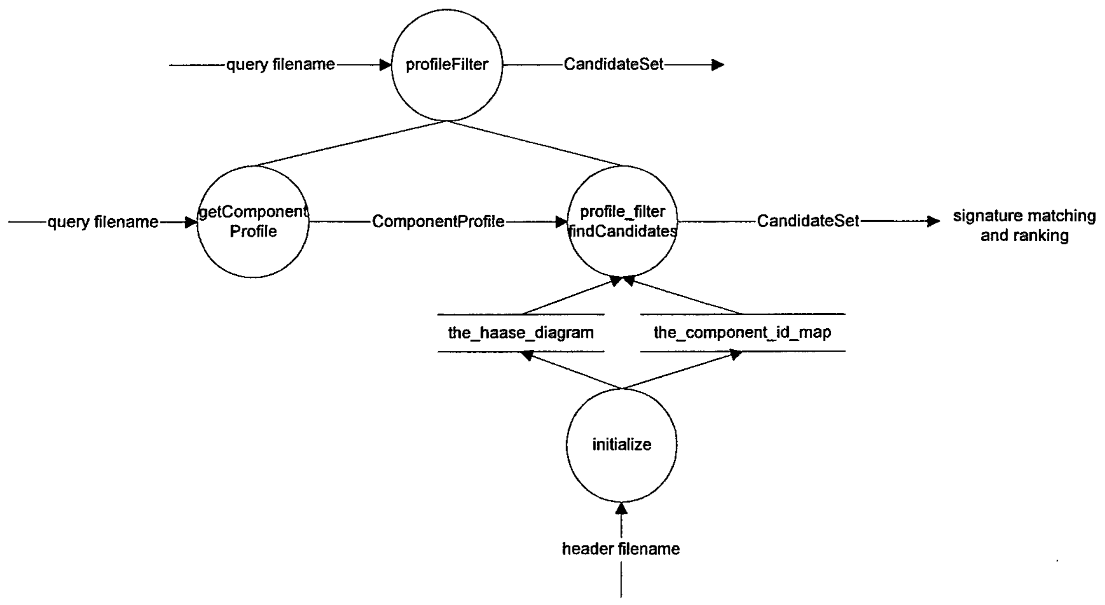


Figure 10: Decomposition of profileFilter

profileFilter decomposes into two main functions: *getComponentProfile* and *findCandidates.* getComponentProfile reads a PSDL specified query, computes its ComponentProfile and passes it to findCandidates where the actual profile filtering takes place. The decomposition of getComponentProfile is shown in Figure 11. getComponentProfile has been designed to take a GenericsMap if the component it is processing is generic. To process queries, which are assumed to NOT be generic however[7], the GenericsMap passed in to getComponentProfile can just be empty.

[7] [9] cites the handling of generic queries as a topic of future research.

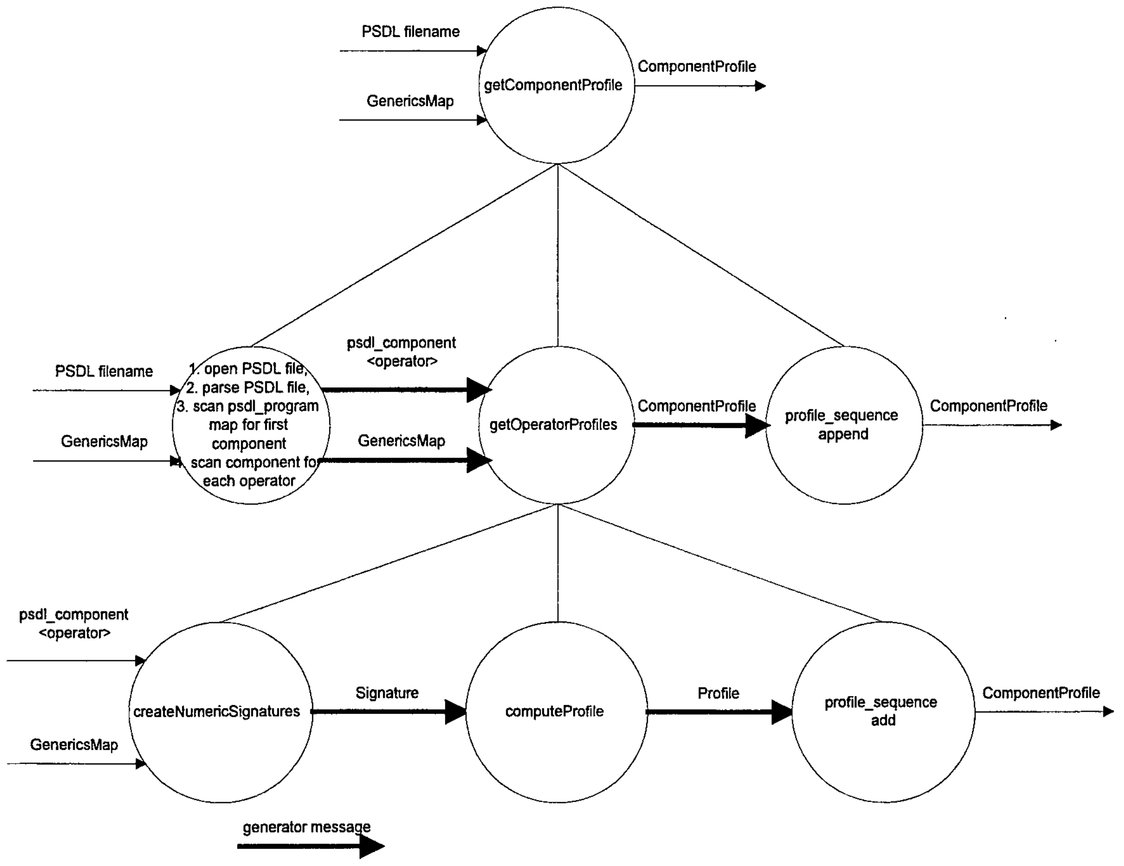


Figure 11: Decomposition of getComponentProfile

The algorithm to compute profiles was given as a class project in [1]. The approach taken by the author's group was to have the algorithm use a language independent (including independence from PSDL) signature for greater reuse potential with other specification languages. The resulting signature, which is referred to as a *numeric signature*, is represented as an array of integers where each unique integer represents a different sort group and each entry in the array indicates to which sort group each argument belongs. Negative integers were used for generic sort groups and the array was terminated with a 0. For example, given *id* from the component listed in Table 6 was a generic parameter, the numeric signatures for each operation would be generated as listed in Table 8.

Table 8: Numeric Signatures for Bag

Component Operations	Numeric Signature
add: id bag → bag	[-1,1,1,0]
merge: bag bag → bag	[1,1,1,0]
member: id bag → bool	[-1,1,2,0]
freq: id bag → natural	[-1,1,2,0]

This format works fine for the profile definition in [9] and even for the value sort frequency improvement presented in section IV.B.1. The problem with this format arises when adding the other profile improvements that are concerned with measuring the frequency of predefined and user-defined sorts. The integers in the numeric signature do not carry with them sort identity. Hence the numeric signature was modified to contain these two profile improvement properties directly. The first integer after the original terminating 0 represents the type sort frequency and the remaining integers represent the frequency of the predefined sort groups. Also, the createNumericSignatures method was modified to take a GenericsMap to create a numeric signature with the generic parameters instantiated and therefore remove any negative integers representing generic parameters. The improvements are represented in Table 9 and assume *id* is mapped to a boolean.

Table 9: Improved Numeric Signatures

Component Operations	Numeric Signature
add: id bag → bag	[1,2,2,0,2,0,0,0,0,0]
merge: bag bag → bag	[1,1,1,0,3,0,0,0,0,0]
member: id bag → bool	[1,2,1,0,1,2,0,0,0,0]
freq: id bag → natural	[1,2,3,0,1,0,0,0,1,0]

With these improvements to the numeric signatures we can now develop an algorithm to compute profiles for generic components with all the improvements presented in this thesis from a language independent format. This algorithm is represented by the function *computeProfile* and its source code can be found in the appendix.

V. SIGNATURE MATCHING

A. CURRENT STATE OF THE ART

[9] proposes a strategy for signature matching that involves the discovery of *Partial Signature Maps.* A partial signature map maps operations and sorts from the query to operations and sorts in the candidate. The signature maps are called partial because it is possible that not all of the query's operations can be mapped to operations in the candidate component. A signature map that successfully maps all of the query's operations is considered a *Full Signature Map.*

In [9] syntactic profiles play an important role in signature matching. Their use in profile filtering eliminate syntactically incompatible components from being passed on to signature matching, but most importantly they provide a quick test for determining which operations in the query and the candidate have the potential for matching. Simply stated, signature matching is only performed on operations that have equal operation profiles.

B. IMPROVEMENTS

Signature matching becomes expensive as the sizes of the query and the candidate grow. More specifically, the number of possible operation pairings grows exponentially as the number of syntactically compatible operations (operations with equal syntactic profiles) increases. To compound the problem, the number of possible sort matches for each pairing grows exponentially as the number sort occurrences increases. These combinatorial explosions can result in large search spaces. Table 10 illustrates the problem:

Table 10

Query	Operation Profiles	Component
Q1: E A D $\rightarrow$ B	[4,1,4,0,0,0,0,0,0]	C1: V W Y $\rightarrow$ Z
Q2: A B C D $\rightarrow$ F	[5,1,5,0,0,0,0,0]	C2: J K L M $\rightarrow$ Y
Q3: D B C A $\rightarrow$ E	[5,1,5,0,0,0,0,0]	C3: W X Y Z $\rightarrow$ T
	[5,1,5,0,0,0,0,0]	C4: W X Y Z $\rightarrow$ S

The query operation Q1 can only match to C1 because C1 is the only operation in the component that contains a compatible operation profile. Q2, however, can match to C2, C3 and C4. Furthermore C3 can also match to C2, C3, and C4. Before sort matching occurs we already have many possible combinations of operation parings to test. The problem really explodes as the sorts for each of these possible pairings undergo the matching process. For the Q2/C2 pairing, A can match to J, K, L or M. For each of these possibilities B must then be matched to the remaining types in C2. This continues until all the possibilities are permuted for the Q2/C2 pairing. And this is just for the Q2/C2 pairing!

Below are several improvements that can be made to combat the combinatorial explosion problems associated with matching large components.

1. Operation Ordering

[9] suggests ordering the operations in the query and components by their syntactic profiles as a possible improvement to signature matching. This would allow the signature matching algorithm to sequentially step through the operations for matching and reduce the number of combinations to be considered.

The signature matching algorithm presented in this thesis uses the concept of operation ordering to help constrain the search by matching *smaller* operations before larger operations. By ordering profiles lexicographically, the smallest operations would be the operations with profiles that come first in the ordering. For example, in Table 10 Q1 is smaller than Q2 because it contains less sort occurrences. This is indicated by the first property in the profile and therefore Q1's profile is ordered before Q2. Given this ordering, we can intelligently match the sorts for smaller operations before matching the sorts for larger operations. This is advantageous because smaller operations constrain the number of matching possibilities and therefore can contribute to a quick reduction of the search space. This is explored in greater detail in the section on design and implementation for signature matching.

Additional techniques for reducing the search space that are not dependant on operation orderings are considered next.

2. Match Outputs

When a query operation is mapped to a candidate operation we can immediately attempt to map the value sorts of the operations because all operations are normalized to the point of having a single output [9][10]. This reduces the search space in two ways. First, if either of the value sorts is already mapped (because of a previous operation mapping) then it is possible the operations cannot be mapped. This would be based simply on the fact that the value parameters are already mapped to *different* sorts. Table 11 illustrates this concept:

Table 11

Query	Operation Profiles	Component
Q1: E A D $\rightarrow$ B	[4,1,4,0,0,0,0,0,0]	C1: V W Y $\rightarrow$ Z
Q2: A B C D $\rightarrow$ B	[5,1,5,0,0,0,0,0]	C2: J K L M $\rightarrow$ Y
Q3: D B C A $\rightarrow$ E	[5,1,5,0,0,0,0,0]	C3: W X Y Z $\rightarrow$ T
	[5,1,5,0,0,0,0,0]	C4: W X Y Z $\rightarrow$ S

Suppose we map Q1 to C1. This would mean B would have to map to Z. Now we move to Q2. Suppose we attempt to map Q2 to C2. This would mean B would have to map to Y but this is illegal because B was already mapped to Z! Thus we can immediately prune this branch of the search space and try to map Q2 to C3, which coincidentally will not work either. Hence we have a way of quickly eliminating possible operation mappings before moving on to the potentially more expensive task of mapping the input sorts.

The second way this technique reduces the search space is by constraining the number of input sorts that have to be matched in an operation. As we saw from the example Table 10 illustrated, the search space grows exponentially as the number of unmapped sorts for an operation grows. Using Table 10 again, if we map Q1 to C1 and Q2 to C2 then by applying the technique of immediately matching the output sort, we would have B mapped to Z and F mapped to Y. The fact that B is mapped means that we can eliminate B from the set of unmapped sorts in Q2 when performing sort matching for the Q2/C2 pair. This means only three of the four input sorts would have to be permuted to discover a sort mapping. It turns out that in this particular case, however, since B is mapped to Z, Z, or some supertype of Z, would have to be present in C2's set of input sorts but it is not, therefore we can eliminate the Q2/C2 pairing immediately and prune this branch from the search space.

3. Match Predefined Types

[9] and [2] both allude to the fact that basic types must be preserved in the partial signature map. Such a rule well serves the quest for reducing the signature matching search space by establishing more constraints that can be applied early in the process. For example, the previous section described how the output parameters could be matched immediately following an operation mapping to determine if such a pairing was worth exploring further. Incompatibilities were not caught, however, until at least two operations had been proposed for matching. By applying the constraints that predefined types impose, we have an opportunity to short circuit the branch even earlier. Consider Table 11 for example. If B is an integer, then Z must belong to the integer sort group such that Z is a subtype of B.[8] If Z does not meet this criteria than the branch can be pruned immediately and Q1 and C1 will never be considered for matching. If Z did pass such constraints then Q2 and C2 can be considered for matching, thereby subjecting Y to the same constraints that Z was required to pass.

The preservation rules of predefined types can also be used to reduce the number of unmapped input sorts to permute. All of the query's predefined sorts can be tested for

[8] [2] explicitly declares subtype matching rules for input and output parameters. Such rules and their applicability to the method of signature matching described in this thesis are addressed in section V.C.3.

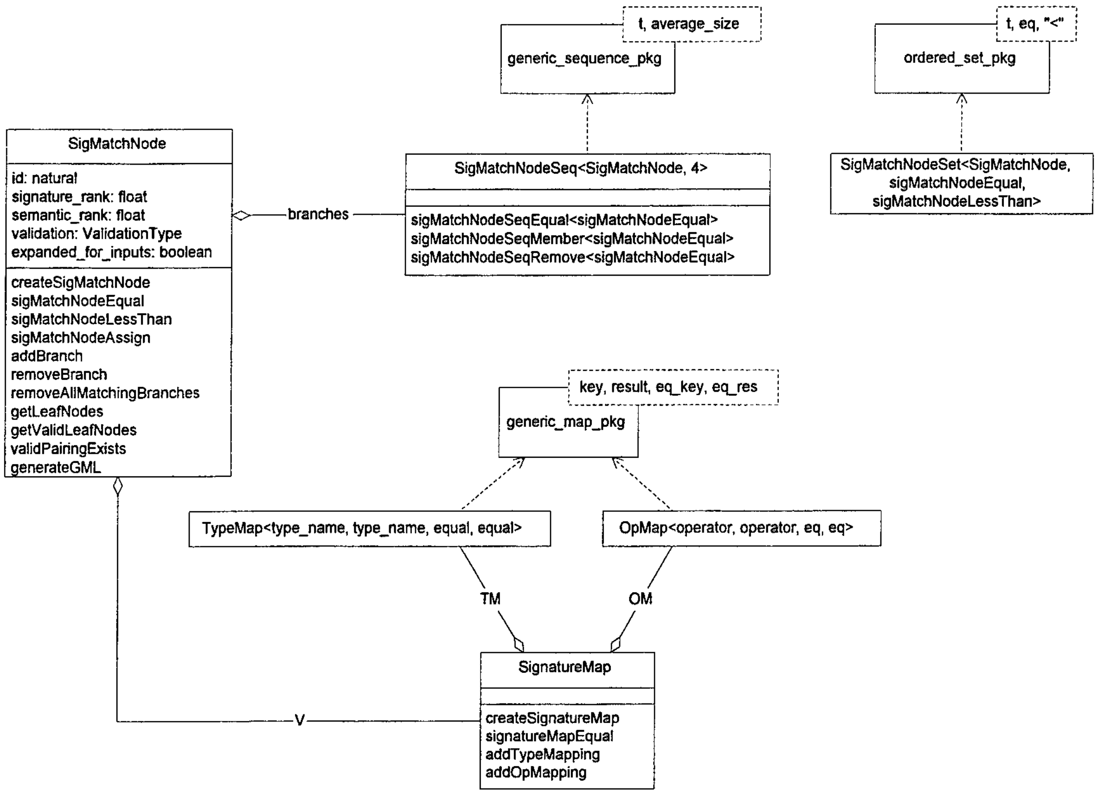
compatibility with the candidate before the permutation process begins. If they all have matches then they can all be removed from the query's set of input parameters, leaving just the unmapped user-defined types for permutation. Clearly this can have profound effects on the number of permutations required to evaluate and therefore pare the search space down significantly.

C. DESIGN AND IMPLEMENTATION

This section is divided into two subsections. The first subsection introduces the objects used to implement the signature matching algorithms and the second subsection discusses the signature matching approach in terms of the objects defined in the first subsection.

1. Signature Matching Types

In order to better illustrate the signature matching strategy proposed in this thesis we must first examine the data types used to carry out the strategy. Figure 12 depicts the signature matching objects.



t, average_size
generic_sequence_pkg
t, eq, "<"
ordered_set_pkg
SigMatchNode
id: natural
signature_rank: float
semantic_rank: float
validation: ValidationType
expanded_for_inputs: boolean
createSigMatchNode
sigMatchNodeEqual
sigMatchNodeLessThan
sigMatchNodeAssign
addBranch
removeBranch
removeAllMatchingBranches
getLeafNodes
getValidLeafNodes
validPairingExists
generateGML
branches
SigMatchNodeSeq<SigMatchNode, 4>
sigMatchNodeSeqEqual<sigMatchNodeEqual>
sigMatchNodeSeqMember<sigMatchNodeEqual>
sigMatchNodeSeqRemove<sigMatchNodeEqual>
SigMatchNodeSet<SigMatchNode, sigMatchNodeEqual, sigMatchNodeLessThan>
key, result, eq_key, eq_res
generic_map_pkg
TypeMap<type_name, type_name, equal, equal>
OpMap<operator, operator, eq, eq>
TM
OM
SignatureMap
createSignatureMap
signatureMapEqual
addTypeMapping
addOpMapping
V

Figure 12: Signature Matching Types

The first object of interest is the *SignatureMap*. This is used to store the operation and type mappings between a query and a candidate and follows [9]'s definition of a partial signature map.

At the crux of the design is the *SigMatchNode*. This data type is used to represent solutions in the signature matching search space by being represented as a node in a tree data structure. The node stores the signature and semantic ranks of the solution (the SignatureMap V) it represents and maintains validation and expansion information for search space maintenance. Since this object is used to form a tree, a handle to a single SigMatchNode can be used to contain the entire search space. When the signature matching process is finished, all the leaves of this tree can be considered valid solutions and therefore can be "clipped" from the tree and returned as the set of solutions. The *getLeafNodes* method is the leaf "clipper" in this case.

Finally, the *SigMatchNodeSet* is used to store the set of SigMatchNodes that will be placed in the Candidate (section IV.C.4) object. Notice first that this collection is a set so that duplicate solution nodes can be easily eliminated and second that this set is ordered. The ordering is defined by the signature rank until semantic matching is performed. Once semantic matching has taken place, the semantic rank takes precedence.

2. Signature Matching Strategy

A high-level view of the signature matching strategy is depicted in Figure 13.

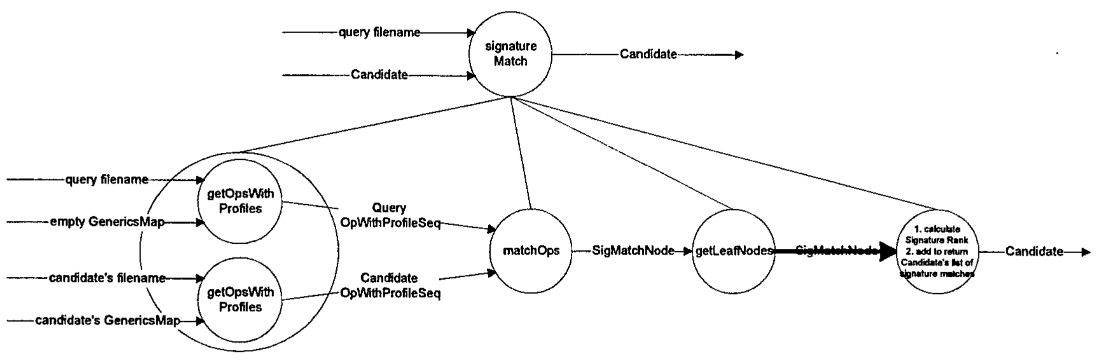
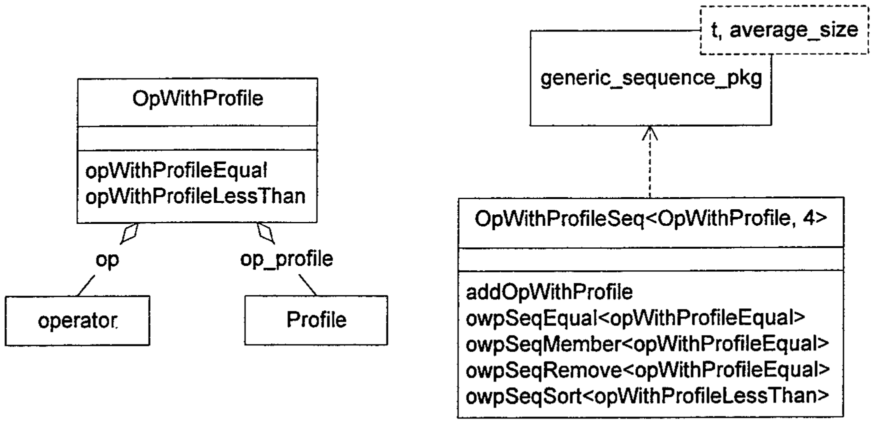


Figure 13: High-level View of Signature Matching

This illustration shows the context in which the core signature matching function, *matchOps*, operates. Once profile filtering is complete, each candidate with a profile rank above a certain threshold is passed to *signatureMatch* along with the original query. signatureMatch then outputs the same candidate passed in but with its set of SigMatchNodes populated.

signatureMatch decomposes into four major steps. The first step calculates the profiles for the operations in the query and the candidate and returns them in the form of *OpWithProfile* sequences. An OpWithProfile, depicted in Figure 14, is simply an association between an operator and its profile. An *OpWithProfileSeq* is a sequence of OpWithProfiles ordered by the lexicographic ordering on profiles used in *opWithProfileLessThan.* The advantages of such an ordering were detailed in section V.B.1. The query's and candidate's OpWithProfileSeq is then passed to matchOps where the actual signature matching takes place. matchOps passes the root SigMatchNode of the entire signature matching search space for the particular query and candidate to getLeafNodes where the valid signature match solutions represented in the leaves of the search space are extracted into a set of SigMatchNodes. Finally, the signature rank for each SigMatchNode in the set is computed and the set is then assigned to the Candidate.



t, average_size
generic_sequence_pkg
OpWithProfile
opWithProfileEqual
opWithProfileLessThan
op
op_profile
operator
Profile
OpWithProfileSeq<OpWithProfile, 4>
addOpWithProfile
owpSeqEqual<opWithProfileEqual>
owpSeqMember<opWithProfileEqual>
owpSeqRemove<opWithProfileEqual>
owpSeqSort<opWithProfileLessThan>

Figure 14: OpWithProfile Data Types

The core signature matching routine is matchOps. Ada-like pseudo-code for matchOps is listed below:

```
procedure match_ops(query: in OpWithProfileSeq, candidate: in OpWithProfileSeq,
        root_sn: in out SigMatchNode) is
    temp_sn, return_val: SigMatchNode;
    temp_query, temp_candidate: OpWithProfileSeq;
    q_op, c_op: OpWithProfile;
begin
    return_val := root_sn;

    --
    -- depth-first-search into possible operation pairings
    --
    temp_query := query;
    temp_candidate := candidate;
    foreach OpWithProfile q_op in query loop
        foreach OpWithProfile c_op in candidate
                where q_op.op_profile = c_op.op_profile loop
            temp_sn := root_sn;
            op_map_pkg.bind(q_op.op, c_op.op, temp_sn.V.OM);
            if not validPairingExists(temp_sn.V.OM, return_val) then
                if match_outputs(temp_sn) then
                    if match_basics(get_basics(q_op.inputs),
                            get_basics(c_op.inputs)) then
                        temp_query := temp_query - q_op;
                        temp_candidate := temp_candidate - c_op;
                        match_ops(temp_query, temp_candidate, temp_sn);
                        addBranch(temp_sn, return_val);
                    end if;
                end if;
            end if;
        end loop;
    end loop;
    --
    -- depth-first-search into possible input pairings
    -- prune until all leaves are valid solutions
    --
    pruned := true;
    while pruned loop
        pruned := false;
        root_sn := return_val;
        foreach leafnode leaf_sn in root_sn loop
            if leaf_sn.validation = UNKNOWN then
                if not match_inputs(leaf_sn) then
                    leaf_sn.validation := INVALID;
                elsif not verify_subtypes(leaf_sn) then
                    leaf_sn.validation := INVALID;
                else
                    if length(leaf_sn.branches) = 0 then
                        leaf_sn.validation := VALID;
                    else
                        leaf_sn.expanded_for_inputs := true;
                    end if;
                end if;
                if leaf_sn.validation = INVALID then
                    removeAllMatchingBranches(leaf_sn, return_val);
                    pruned := true;
                end if;
            end if;
        end loop;
    end loop;

    root_sn := return_val;
end match_ops;
```

There are two main sections to this procedure. The first is a depth-first search into the space of all compatible operation pairs and the second is a combined matching of input sorts and retraction of invalid nodes. The first section steps through each operation in the query, trying to match it to an operation in the candidate. This is done by invoking the following three steps in order: first verifying that the profiles are equal, second verifying if the outputs match (see section V.B.2) and third verifying that the predefined types can match (see section V.B.3). If any of these three steps fail, the operation pairing is not considered and the remaining tests are immediately short-circuited to reduce time. If the three steps succeed then the pairing is added as a branch to the root SigMatchNode passed in to matchOps and matchOps is recursively called again with the same query and candidate OpWithProfileSeqs with the operations just paired removed. Figure 15 illustrates the search space for possible operation pairings for the query and component in Table 10. The highlighted path is the path searched before moving on to the second part of matchOps that involves matching the input sorts and performing any possible retractions of invalid nodes. The rest of the space is depicted here to illustrate the nature of the search space but at this point only the highlighted nodes have been instantiated.

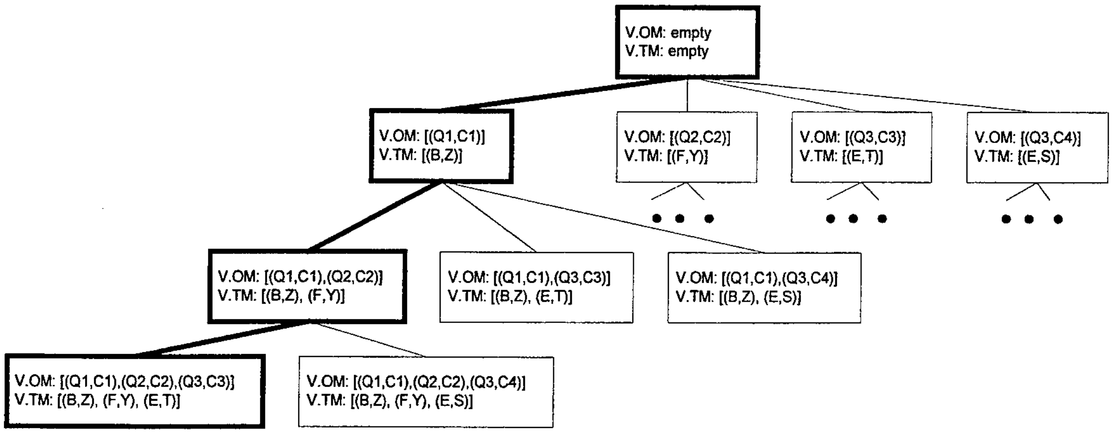


Figure 15: Example Search Space for Compatible Operations

Now that we have searched to the bottom of this particular path we can now begin expanding the search space for matching the inputs for the various operations paired in the node. Ada-like psuedo-code is listed below for this phase of signature matching.

```
--
-- Function: match_inputs
--
function match_inputs(root_sn: in out SolutionNode) return boolean is


    function match(q_inputs: in TypeSequence; c_inputs: in TypeSequence;
          root_sn: in out SolutionNode) return boolean is
    begin
        -- recursive stopping case
        if size(q_inputs) = 0 then
            return;
        end if;

        return_val := root_sn;

        new_q_inputs := q_inputs;
        new_c_inputs := c_inputs;

        -- verify mapped inputs in q_inputs are legally mapped
        -- and set new_q_inputs and new_c_inputs to only
        -- the unmapped inputs
        foreach input_type qi in q_inputs loop
            if type_map_pkg.member(qi, root_sn.V.TM) then
                ci := type_map_pkg.fetch(root_sn.V.TM, qi);
                -- if the current input type is already mapped
                -- then make sure it is mapped to an existing type
                -- in the candidate's input.
                if not type_sequence_pkg.member(ci, c_inputs) then
                    return false;
                end if;
                new_q_inputs := new_q_inputs - qi;
                new_c_inputs := new_c_inputs - ci;
            end if;
        end loop;

        qi := q_inputs[1];
        foreach input_type ci in c_inputs loop
            temp_sn := root_sn;
            temp_sn.expanded_for_inputs := false;
```

```
                type_map_pkg.bind(qi, ci, temp_sn.V.TM);
                if not match(new_q_inputs-qi, new_c_inputs-ci, temp_sn) then
                    return false;
                end if;
                addBranch(temp_sn, return_val);
            end loop;

            root_sn := return_val;
            return true;
        end match;

    begin
        foreach op_mapping om in root_sn.V.OM loop
            -- remove the input types that have already been mapped
            q_inputs := om.key.inputs - type_map_pkg.map_domain(root_sn.V.TM);
            c_inputs := om.result.inputs - type_map_pkg.map_range(root_sn.V.TM);

            -- if the number of remaining input types for the query and
            -- the candidate are unequal than the operations cannot match
            if type_sequence_pkg.length(q_inputs) /=
                    type_sequence_pkg.length(c_inputs) then
                return false;
            end if;

            -- if the node has already been expanded before to try and match
            -- the inputs and it still has unmapped input types then return
            -- false so we won't try again
            if root_sn.expanded_for_inputs then
                return size(q_inputs) = 0;
            else
                return match(get_basics(q_inputs), get_basics(c_inputs), root_sn);
            end if;
        end loop;
    end match_inputs;
```

This function first removes any sorts from the set of inputs that have already been mapped. At this point in our example, this would mean removing any input sorts that were the same as the output sorts since the output sorts have been mapped. If this results in an uneven number of unmapped sorts between the query and the candidate then we can immediately stop and return false since there cannot be a match. Next, if this node has already been expanded in the past and it ultimately led to an invalid node then we do not want to expand this node again since we know where it leads. Finally, if we make it through these preliminary checks then we can pass the node on to the recursive function *match* that will expand the node into all the possible input sort pairings to be investigated. If there are no legal possibilities, match will return false and cause match_inputs to return false, signaling match_ops to flag this node as invalid.

Going back to our example, the node for which we are currently trying to match inputs cannot be expanded because Q1's first input sort, E, is already mapped to T but there is no T in the input sorts for C1! The test for making sure the number of unmapped input sorts in the query is equal to the number of unmapped input sorts for the candidate will fail and cause match_inputs to return false. Looking back at match_ops, this will cause the node to be pruned and match_ops will pop back to its previous instantiation on the stack and search the next possibility in the search space. The tree at this point is shown in Figure 16.

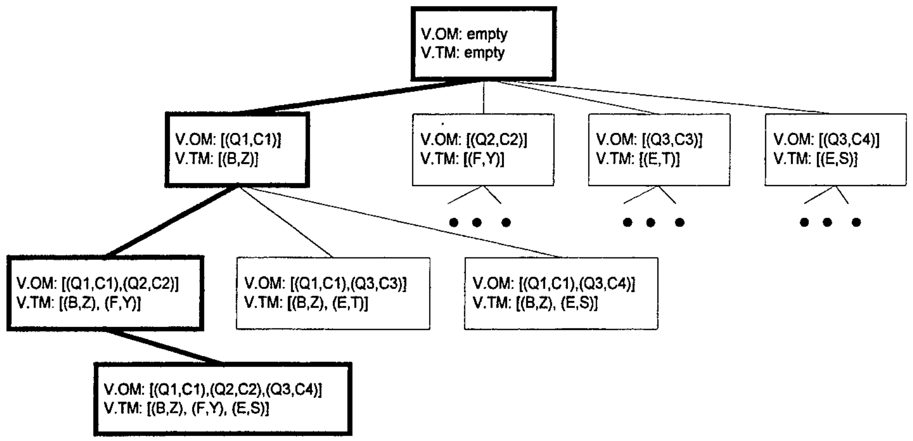


Figure 16: Search Space after First Pruning

The current node will be pruned for the same reasons the first node was pruned: E is already mapped to a sort that does not exist in the candidate's input sorts. So again, another node is pruned, match_ops pops back up and we are now left with the tree in Figure 17.

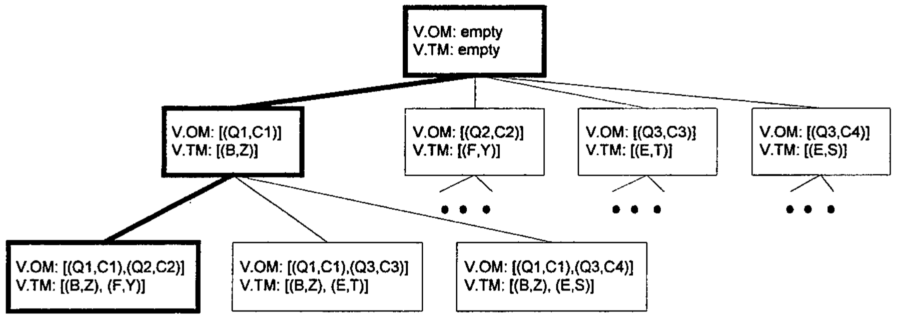


Figure 17: Search Space after Second Pruning

Now we have a node that will pass all the preliminary steps and successfully expand for all the possible input pairings. Given the relatively large number of unmapped inputs in Q1 and Q2, the expansion is quite significant. Figure 18 shows part of this expansion.

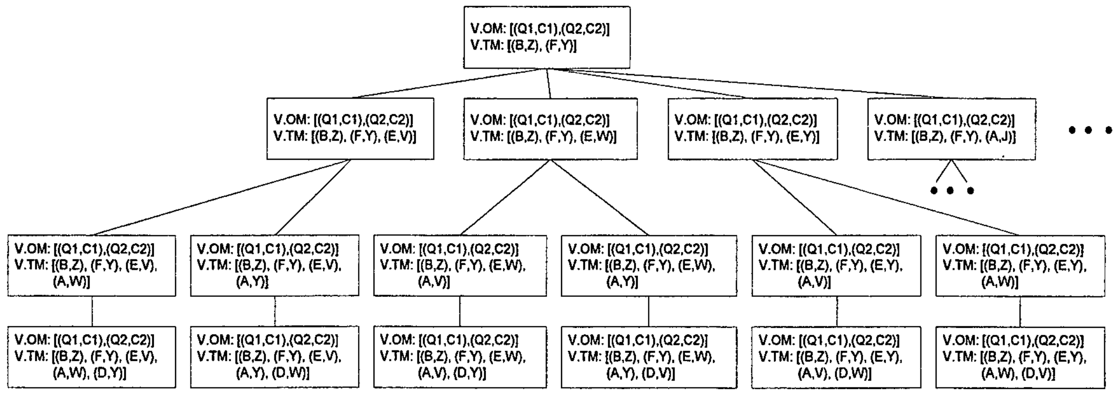

V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,V)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,W)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,Y)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (A,J)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,V), (A,W)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,V), (A,Y)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,W), (A,V)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,W), (A,Y)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,Y), (A,V)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,Y), (A,W)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,V), (A,W), (D,Y)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,V), (A,Y), (D,W)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,W), (A,V), (D,Y)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,W), (A,Y), (D,V)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,Y), (A,V), (D,W)]
V.OM: [(Q1,C1),(Q2,C2)] V.TM: [(B,Z), (F,Y), (E,Y), (A,W), (D,V)]

Figure 18: Input Sort Matching Expansion

The figure effectively illustrates the combinatorial explosion associated with large numbers of unmapped inputs and underscores the need for the improvements cited in section V.B. Figure 18 only shows the expansion for the first pairing Q1/C1 and *one* node for the pairing Q2/C2. Q2/C2 is expanded in the same way Q1/C1 is expanded but is not shown here due to space limitations. The entire expansion is ultimately returned back to match_ops where the valid leaves will continue to go through the match_inputs expansion. In this case all the leaves shown in Figure 18 will have to be expanded further to add any more mappings Q2/C2 brings on top of the Q1/C1 mappings. Similarly this would be done for Q1/C1 on top of the Q2/C2 mappings in the portion of the tree not shown. The expansion/prune loop in match_ops continues until there are no more leaves to expand and all of the existing leaves are valid.

A legitimate concern might arise from the example thus far regarding the fact that Q1/C1 and Q2/C2 are expanded twice, but in different order. This must take place because it is possible to have different input sort mappings depending on which order the operations are expanded in. For instance, notice in the first node representing Q2/C2's input matching (whose expansion is not shown further in Figure 18 due to space) that A is mapped to J. In the Q1/C1 expansions A is never mapped to J. This possibility would never be explored in this part of the search space if the different ordering of pairing expansions were not represented.

3. Subtype Matching

In section V.C.2 we see that match_ops tests if the predefined types in the operations for an operation pairing can be legally matched. The Ada-like pseudo-code for determining this legality for the input types is listed below.

```
function match_basics(q_basics: in out TypeSequence; c_basics: in out TypeSequence)
      return boolean is
begin
    -- cannot match if query has different number of basics than the
    -- candidate
    if size(q_basics) /= size(c_basics) then
        return false;
    end if;

    --
    -- Basic types: either they must match exactly
    -- or the query's input type must be a
    -- subtype of the component's input type.
    --

    --
    -- filter out the basics that match exactly
    --
    new_q_basics := q_basics;
    new_c_basics := c_basics;
    foreach input_type qi in q_basics loop
        foreach input_type ci in c_basics loop
            if equal(qi, ci) then
                new_q_basics := new_q_basics - qi;
                new_c_basics := new_c_basics - ci;
                break; -- out of inner foreach loop
            end if;
        end loop;
    end loop;
    q_basics := new_q_basics;
    c_basics := new_c_basics;

    --
    -- Filter out the remaining basics that can match to supertypes.
    -- This is done by temporally mapping each query input types to a
    -- supertype in the candidate that is closest in the partial ordering.
    --
    foreach input_type qi in q_basics loop
        foreach input_type ci in c_basics loop
            if subtype_of(qi, ci) then
                found_ci2 := false;
                foreach input_type ci2 in new_c_basics loop
                    if not found_ci2 and subtype_of(qi, ci2) and not equal(ci, ci2)
                            and subtype_of(ci2, ci) then
                        found_ci2 := true;
                    end if;
                end loop;
                if not found_ci2 then
                    new_q_basics := new_q_basics - qi;
                    new_c_basics := new_c_basics - ci;
                end if;
            end if;
        end loop;
        end loop;

        --
        -- if there are any basics left over than match is not possible
        -- since basics cannot be matched to non-basics
        --
        return type_sequence_pkg.length(new_q_basics) = 0;
    end match_basics;
```

Of particular note is the portion of the function that is responsible for subtype matching. Without subtype matching the operation pairing would fail if there were predefined types left over after all the predefined types that can find exact matches were mapped. With subtype matching, however, the possibilities of mapping the remaining predefined types are explored.

[2] defines subtype matching rules for mapping the input and output types of an operation. These rules are summarized below:

1. an input type of a query must be a subtype of the input type in the candidate to which it is mapped
2. an output type of a query must be supertype of the output type in the candidate to which it is mapped

These rules are followed in the pseudo-code above. An interesting case arises when there is more than one supertype available in the candidate. In such a case the algorithm above will choose the supertype closest in the partial ordering for that particular sort group. For instance, if we are trying to map a positive in the query and the candidate has a natural and an integer still unmapped, then the natural is selected over the integer because the positive is closer to the natural in the partial ordering of the integer sort group. This has the advantage that the less refined sorts remain available in the candidate for potential mappings with less refined sorts in the query.

[9] extends [2]'s rules by maintaining subtype consistency throughout the partial signature map. For example, suppose we are trying to match Q1 and C1 from Table 10 and we map E to V and A to W. The subtype rules in [9] state that if E is a subtype of A then V must be a subtype of W, otherwise the mapping is invalid. The test of such consistency in the approach outlined in this thesis is made by the call to *verify_subtypes*

in match_ops. If the test fails for all the mapped sorts in the node then the node is considered invalid and pruned.

VI. EXPERIMENTATION

To test the effectiveness of the syntactic matching improvements presented in this thesis the CAPS software base [10] was selected as well as the following four queries:

Query: Stack

```
TYPE Stack
SPECIFICATION
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Stack : Stack,
      To_The_Stack : Stack
    OUTPUT
      To_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Push
  SPECIFICATION
    INPUT
      The_Integer : Integer,
      On_The_Stack : Stack
    OUTPUT
      On_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Stack,
      Right : Stack
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Depth_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Top_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Integer
    EXCEPTIONS
      Overflow, Underflow
  END

END
IMPLEMENTATION ADA
Stack_Sequential_Bounded_Managed_Iterator
END
```

Query: Set

```
TYPE Set
SPECIFICATION
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set,
Item_Is_Not_In_Set
  END
  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set,
Item_Is_Not_In_Set
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set,
Item_Is_Not_In_Set
  END

  OPERATOR Remove
  SPECIFICATION
    INPUT
      The_Item : Item,
      From_The_Set : Set
    OUTPUT
      From_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set,
Item_Is_Not_In_Set
  END

  OPERATOR Union
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set,
Item_Is_Not_In_Set
  END

  OPERATOR Intersection
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set,
Item_Is_Not_In_Set
  END

  OPERATOR Difference
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set,
Item_Is_Not_In_Set
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set,
Item_Is_Not_In_Set
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_In_Set,
Item_Is_Not_In_Set
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set,
Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Member
  SPECIFICATION
    INPUT
      The_Item : Item,
      Of_The_Set : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set,
Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Subset
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set,
Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Proper_Subset
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set,
Item_Is_Not_In_Set
  END

END
IMPLEMENTATION ADA
Set_Simple_Sequential_Bounded_Managed_Ite
rator
END
```

Query: Map

```
TYPE Map
SPECIFICATION
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Map : Map,
      To_The_Map : Map
    OUTPUT
      To_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound,
Multiple_Binding
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound,
Multiple_Binding
  END

  OPERATOR Bind
  SPECIFICATION
    INPUT
      The_Domain : Natural,
      And_The_Range : Ranges,
      In_The_Map : Map
    OUTPUT
      In_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound,
Multiple_Binding
  END

  OPERATOR Unbind
  SPECIFICATION
    INPUT
      The_Domain : Natural,
      In_The_Map : Map
    OUTPUT
      In_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound,
Multiple_Binding
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Map,
      Right : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound,
Multiple_Binding
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound,
Multiple_Binding
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound,
Multiple_Binding
  END

  OPERATOR Is_Bound
  SPECIFICATION
    INPUT
      The_Domain : Natural,
      In_The_Map : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound,
Multiple_Binding
  END

  OPERATOR Range_Of
  SPECIFICATION
    INPUT
      The_Domain : Natural,
      In_The_Map : Map
    OUTPUT
      Result : Ranges
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound,
Multiple_Binding
  END

END
IMPLEMENTATION ADA
Map_Simple_Noncached_Sequential_Unbounded
_Managed_Iterator
END
```

Query: Queue

```
TYPE Queue
SPECIFICATION
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
    To_The_Queue : Queue
  OUTPUT
    To_The_Queue : Queue
  EXCEPTIONS
    Overflow, Underflow
END

OPERATOR Clear
SPECIFICATION
  INPUT
    The_Queue : Queue
  OUTPUT
    The_Queue : Queue
  EXCEPTIONS
    Overflow, Underflow
END

OPERATOR Add
SPECIFICATION
  INPUT
    The_Item : Item,
    To_The_Queue : Queue
  OUTPUT
    To_The_Queue : Queue
  EXCEPTIONS
    Overflow, Underflow
END

OPERATOR Pop
SPECIFICATION
  INPUT
    The_Queue : Queue
  OUTPUT
    The_Queue : Queue
  EXCEPTIONS
    Overflow, Underflow
END

OPERATOR Is_Equal
SPECIFICATION
  INPUT
    Left : Queue,
    Right : Queue
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Underflow
END

OPERATOR Length_Of
SPECIFICATION
  INPUT
    The_Queue : Queue
  OUTPUT
    Result : Natural
  EXCEPTIONS
    Overflow, Underflow
END

OPERATOR Is_Empty
SPECIFICATION
  INPUT
    The_Queue : Queue
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Underflow
END

OPERATOR Front_Of
SPECIFICATION
  INPUT
    The_Queue : Queue
  OUTPUT
    Result : Item
  EXCEPTIONS
    Overflow, Underflow
END

END
IMPLEMENTATION ADA
Queue_Nonpriority_Nonbalking_Sequential_B
ounded_Managed_Iterator
END
```

The queries are instantiations of generic components from the software base to ensure interesting matching activity. An attempt was made to instantiate the generic parameters differently in order to facilitate observation of the different manifestations of improvement in this thesis. For instance, the *stack* and *queue* query use predefined types for the generic parameters, whereas the *set* and *map* queries do not. As a result, the sensitivity to improvements involving predefined types will be different amongst the queries.

A. GENERIC COMPONENT INSTANTIATION

The CAPS software base contained 80 components, most of which are generic abstract data types. After instantiating all of the generic components with all possible combinations (see section IV.C) the number of searchable components increased to 566. This is a substantial increase and demonstrates the need for quick filters early in the search process.

B. PROFILE FILTERING

1. Software Base Resolution

As mentioned in section IV.B.1, increasing the resolution of profiles will increase the resolution of the software base by requiring more non-empty partitions (haase-nodes) to store the components. An increased partition count means there will be fewer components sharing partitions and therefore contributes to an increase in precision without a loss in recall during profile filtering. A simple metric for determining the effectiveness of the resolution improvements is the number of partitions necessary to store all the components in the software base. The more partitions, the more effective the profile resolution improvement. Figure 19 illustrates the effectiveness of the resolution improvements outlined in this thesis on the CAPS software base. The graph shows that applying all the resolution improvements yields a 65% gain in the number of partitions over no improvements at all.

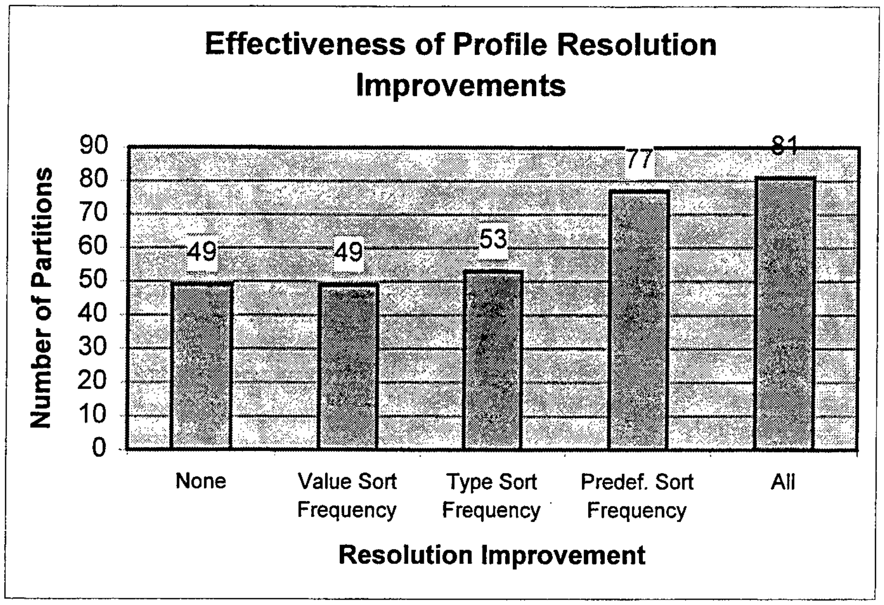


Figure 19: Effectiveness of Profile Resolution Improvements on Software Base Partitioning

By observing the effects of the different resolution improvements individually it is evident that the kind of components in the software base drive the effectiveness of the various improvements differently. For instance, in the CAPS software base the components have similar operations to one another that do not vary in the frequency of the value sort. Thus the value sort frequency improvement has no effect. The components do, however, make reference to various predefined types thereby causing the substantial increase in partitions. If the CAPS software base did not use predefined types at all, however, then such an improvement would obviously have no effect.

2. Profile Filtering Performance

Increasing the resolution of the profiles should cause an increase in precision without a recall penalty. Hence, we want to see a reduction in the number of components returned at high profile rank thresholds. Such behavior is exactly what we see in the graphs illustrated in Figure 20 through Figure 23. For each query a substantially greater number of components are filtered out at high profile rank thresholds when all of the profile improvements are employed then when none of the improvements are employed.

The effectiveness of each profile improvement individually is again dependent upon the properties of the query and the components in software base. As we saw in Figure 19, the CAPS software base is rather sensitive to the predefined sort frequency improvement. As expected, this sensitivity is evident during profile filtering. For example, when the profile rank threshold is set at 1 (requiring a 100% match) the predefined sort frequency causes the number of recalled components to be drastically reduced.

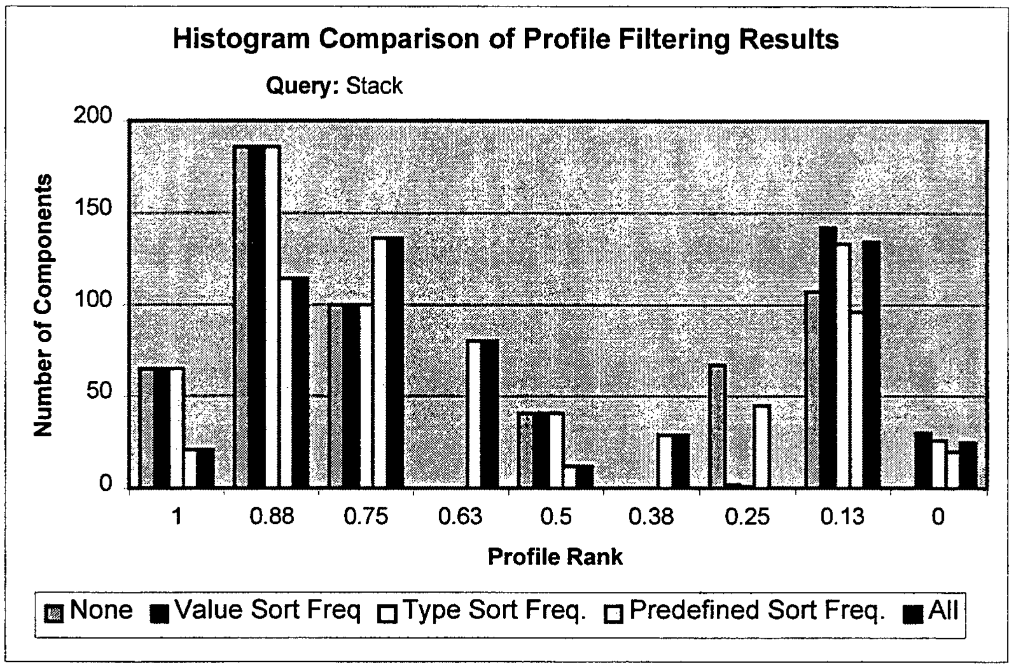


Figure 20: Histogram Comparison of Profile Filtering Results with Stack Query

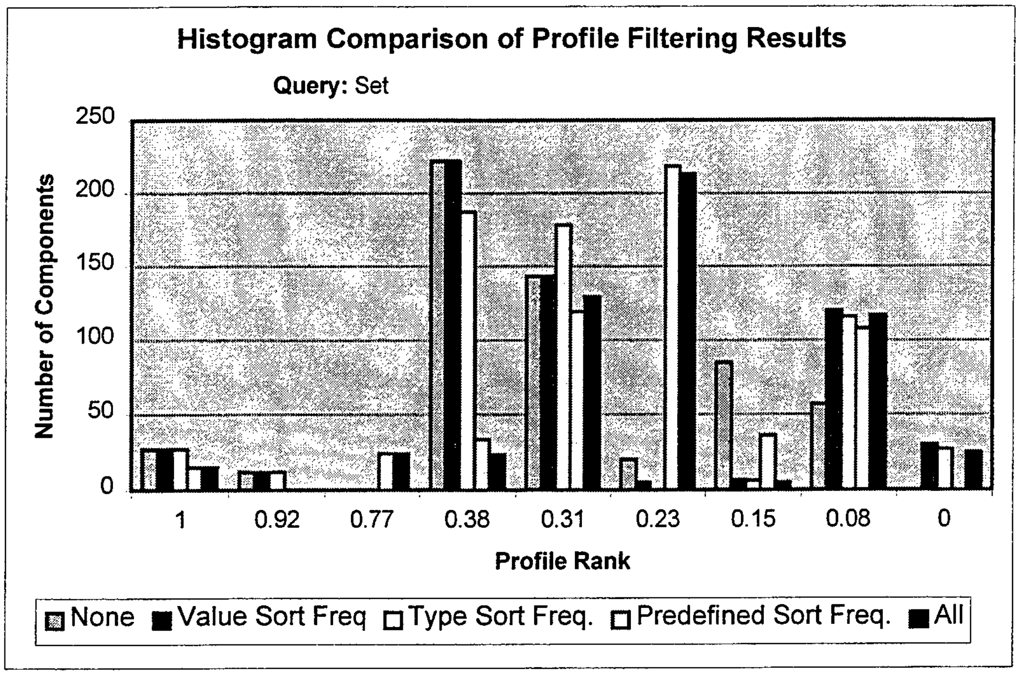


Figure 21: Histogram Comparison of Profile Filtering Results with Set Query

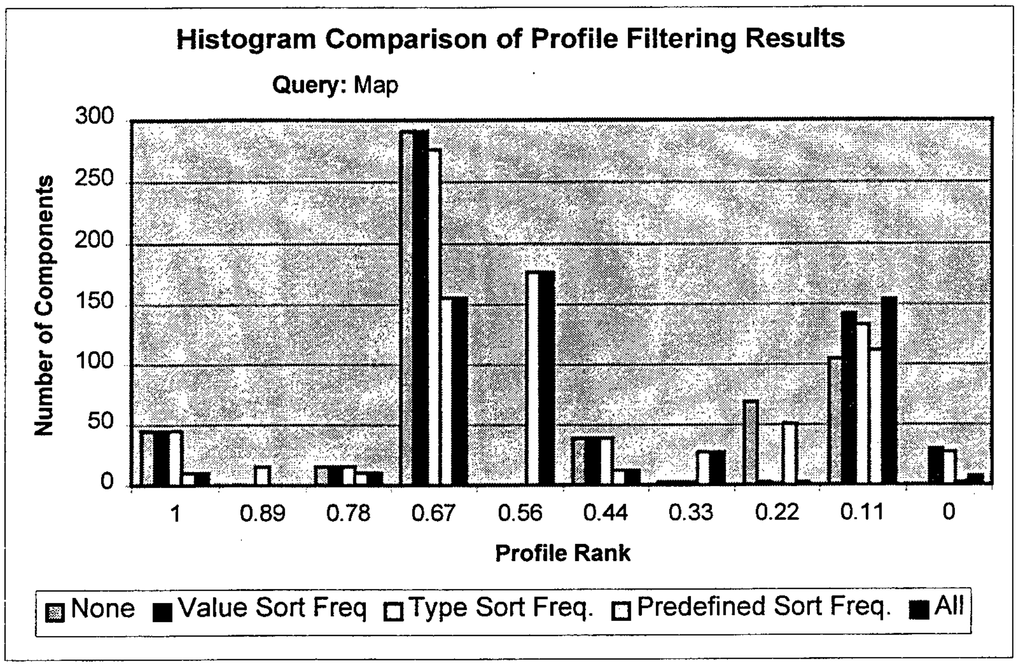


Figure 22: Histogram Comparison of Profile Filtering Results with Map Query

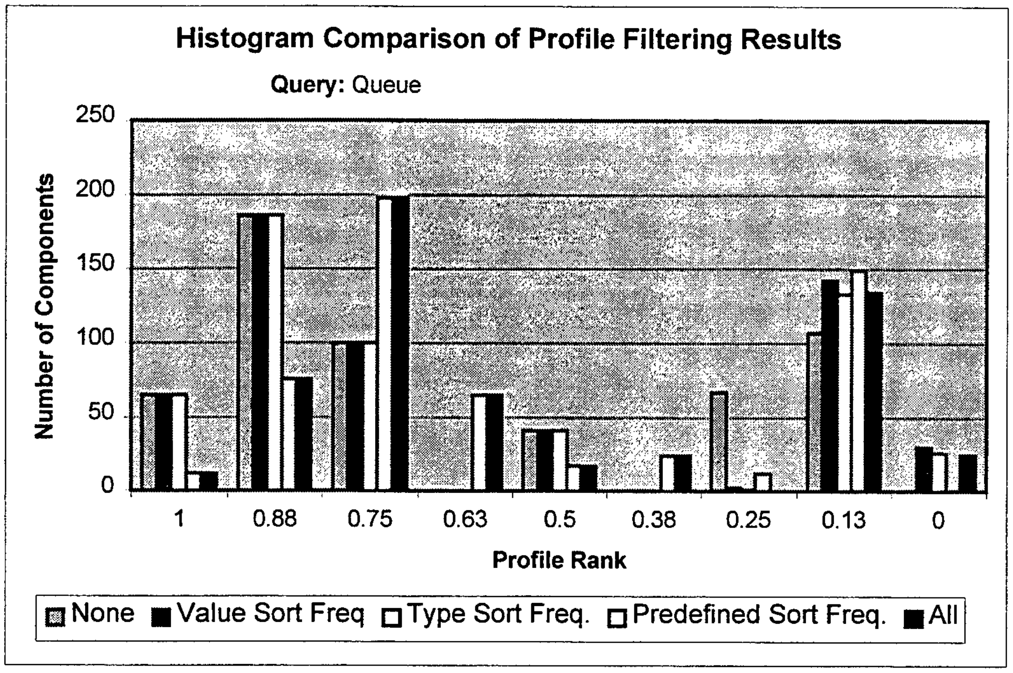


Figure 23: Histogram Comparison of Profile Filtering Results with Queue Query

Figure 24 through Figure 27 present a different perspective on the effectiveness of the resolution improvements during profile filtering. These graphs maintain a running sum of the number of recalled components throughout the continuum of profile rank thresholds. They show us that at a profile rank threshold of .65 (65% of the operations in the query must be in the component) the improvements lose their advantage. In other words, if the user sets the profile rank threshold above .65, the resolution improvements presented in this thesis will have a significantly positive effect on increasing precision.

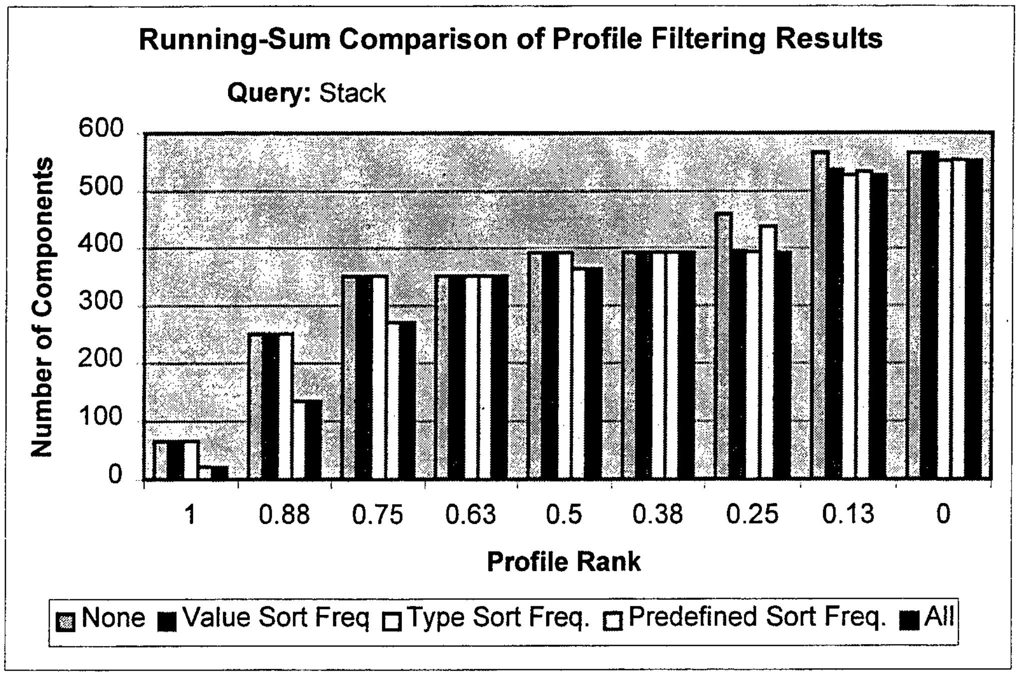


Figure 24: Running-Sum Comparison of Profile Filtering with Stack Query

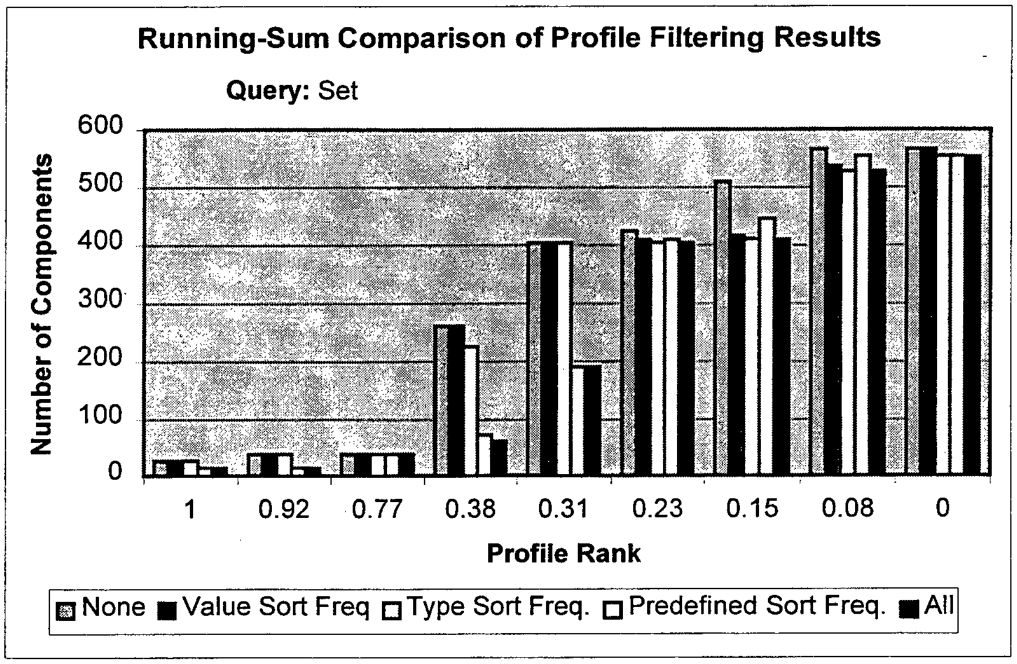


Figure 25: Running-Sum Comparison of Profile Filtering with Set Query

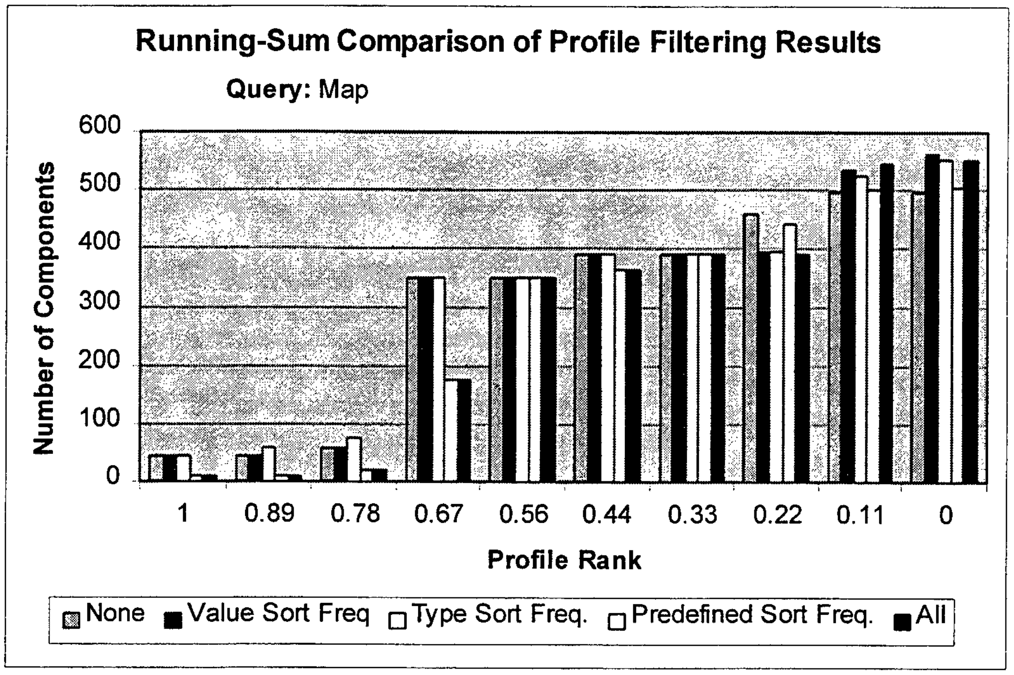


Figure 26: Running-Sum Comparison of Profile Filtering with Map Query

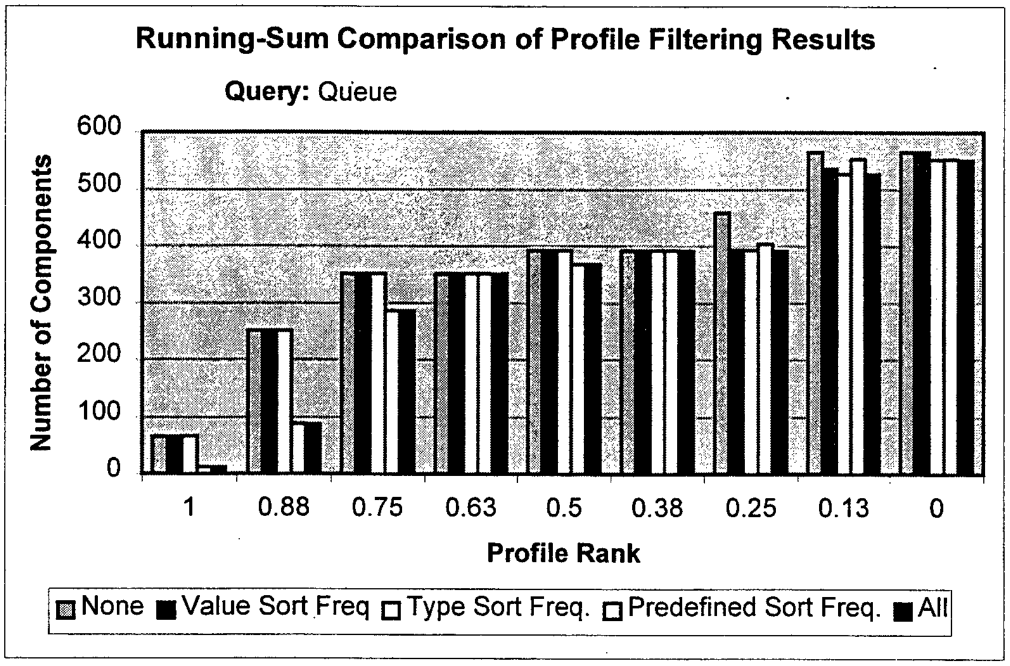


Figure 27: Running-Sum Comparison of Profile Filtering with Queue Query

C. SIGNATURE MATCHING

This thesis has presented improvements to signature matching both indirectly through improvements to profile resolution and directly through early pruning of the search space. This section illustrates the effectiveness of these improvements respectively.

1. Effectiveness of Profile Improvements on Signature Matching

Determining the effects profile resolution improvements have on signature matching is difficult because the traditional rankings that order the outcome of profile filtering and signature matching are not orthogonal. Furthermore, it is difficult to compare the effects the individual profile resolution improvements have on signature matching because they will cause the profile filtering process to potentially return different sets of components to pass on to signature matching. Finally, different queries can cause behavior that is difficult to correlate. To this end, a concise quantification of the effectiveness profile improvements have on signature matching will not be made in this thesis. Rather, informal comments can be made regarding the results of the signature matching process for the four queries and the various resolution improvements in Figure 28 through Figure 31. These graphs compare the effects the different profile resolution improvements have on signature matching by showing distributions of the number of valid partial signature maps for each signature rank. The signature matching was performed on a randomly selected component that had a profile rank of 1 (100% of the query's operations had compatible operation profiles in the component), meaning a signature rank of 1 was possible.

To begin, Figure 28 shows that when all of the profile resolution improvements are active, a valid signature map where 88% of the *stack* query's operations are mapped can be obtained. For the particular candidate chosen, this is not possible when no improvements are active.

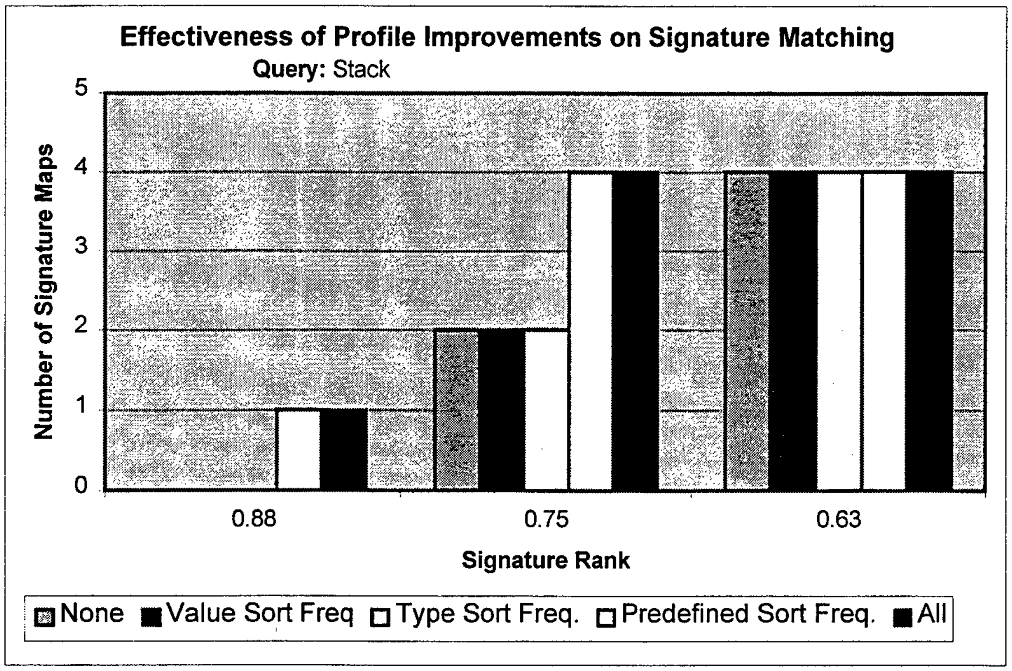


Figure 28: Effectiveness of Profile Improvements on Signature Matching with Stack Query

In Figure 29 two main characteristics are visible. First, the number of valid signature maps that are generated is substantially more than with the other queries. This is due to the fact that the *set* query has many operations that are compatible with the candidate from the software base. As described in section V.B, the possible permutations grow exponentially as the number of operations with compatible operation profiles grows. The second characteristic to notice is the lack of performance from the predefined sort frequency improvement. Throughout the distribution it returns the same number of maps as the version without improvements. This can be attributed to the lack of predefined types in the inputs of the operation signatures, causing the predefined sort frequency improvement to make only a small precision improvement over no resolution improvements for a profile rank of 1 (see Figure 25). The same candidate happened to be chosen for both cases and hence the same signature matching results ensued.

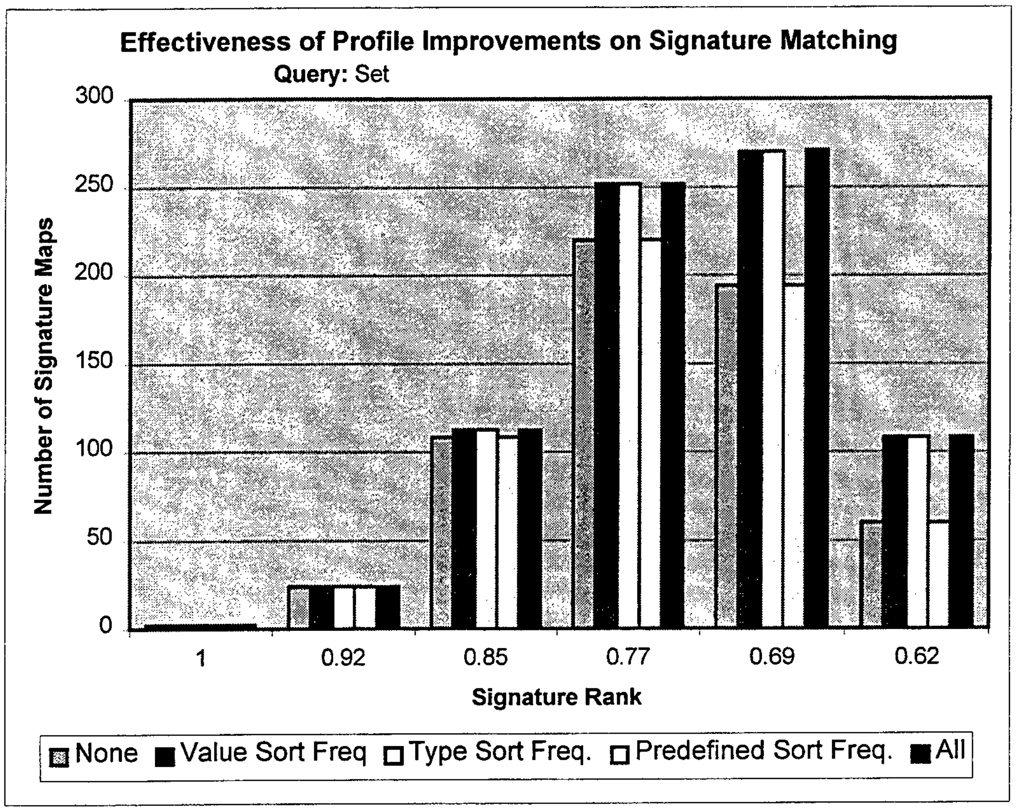


Figure 29: Effectiveness of Profile Improvements on Signature Matching with Set Query

Figure 30 and Figure 31 both have examples of 100% success in syntactic matching. The profile resolution improvements do not make any difference in these examples, however, primarily because the combination of the query and software base caused the random selection of the candidate to select the same candidate.

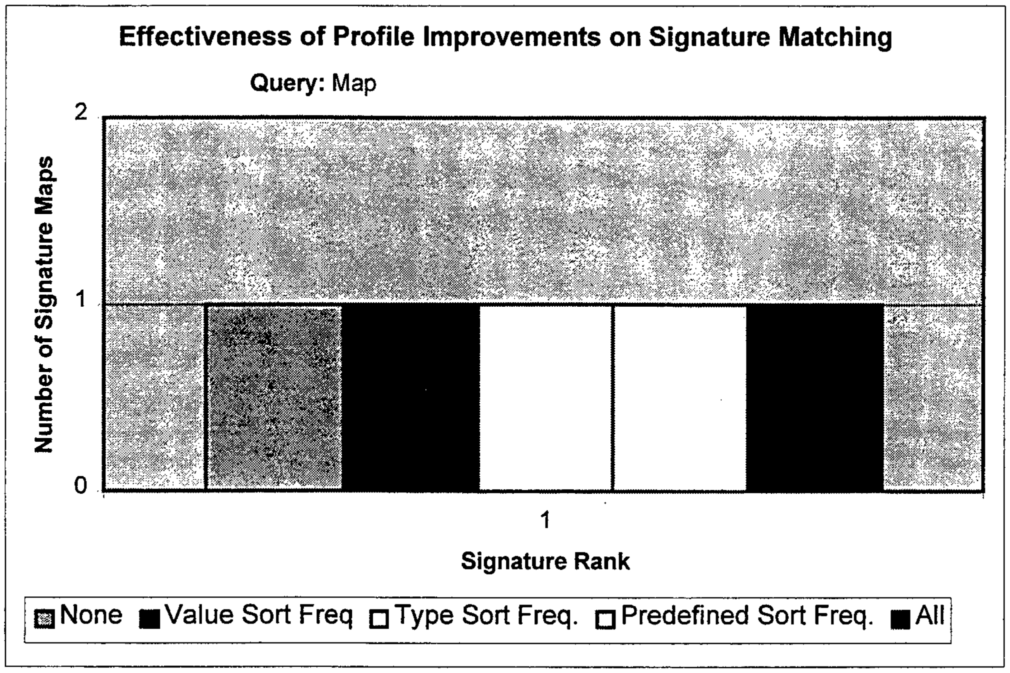


Figure 30: Effectiveness of Profile Improvements on Signature Matching with Map Query

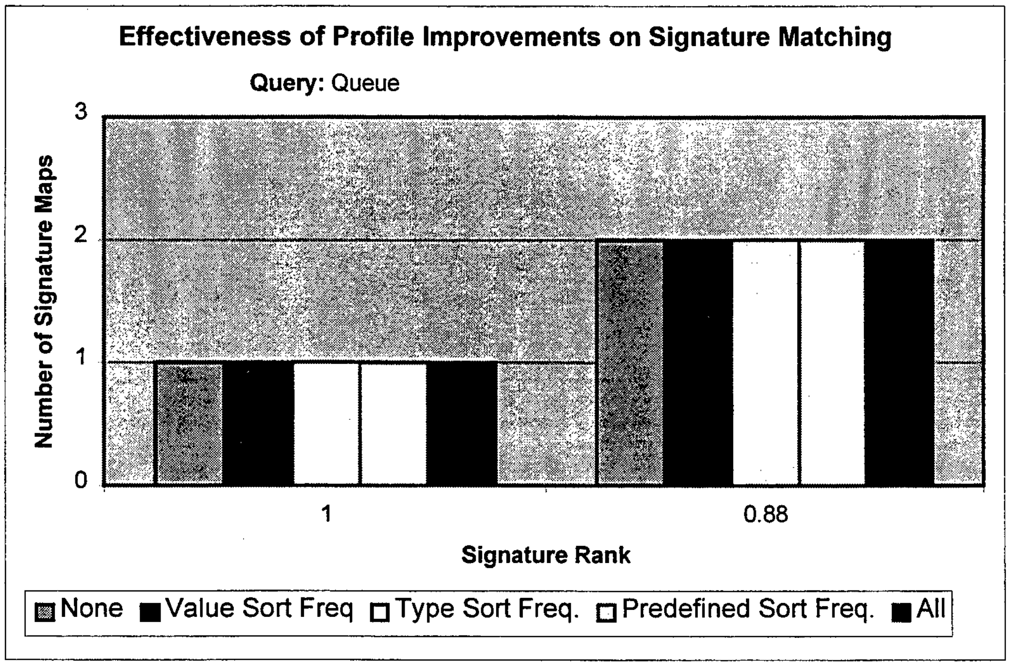


Figure 31: Effectiveness of Profile Improvements on Signature Matching with Queue Query

2. Signature Matching Algorithm Performance

The signature matching improvements presented in section V.B can be observed by counting the number of nodes that pass and fail the early tests for output matching and predefined type matching. Of particular interest is the number of failed nodes. Failed nodes represent nodes that are pruned. Clearly, the more nodes pruned the better. Such a measurement shows off the signature matching improvements presented in this thesis. The graphs in Figure 32 through Figure 35 show pass/fail node measurements for each query and compare between the various profile resolution improvements.

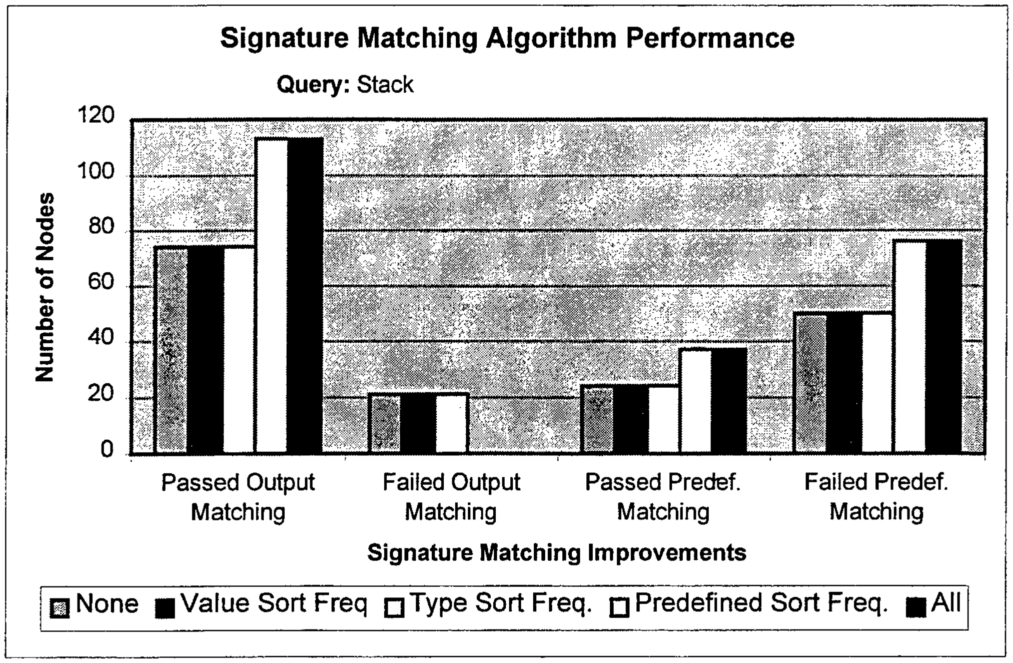


Figure 32: Signature Matching Algorithm Performance with Stack Query

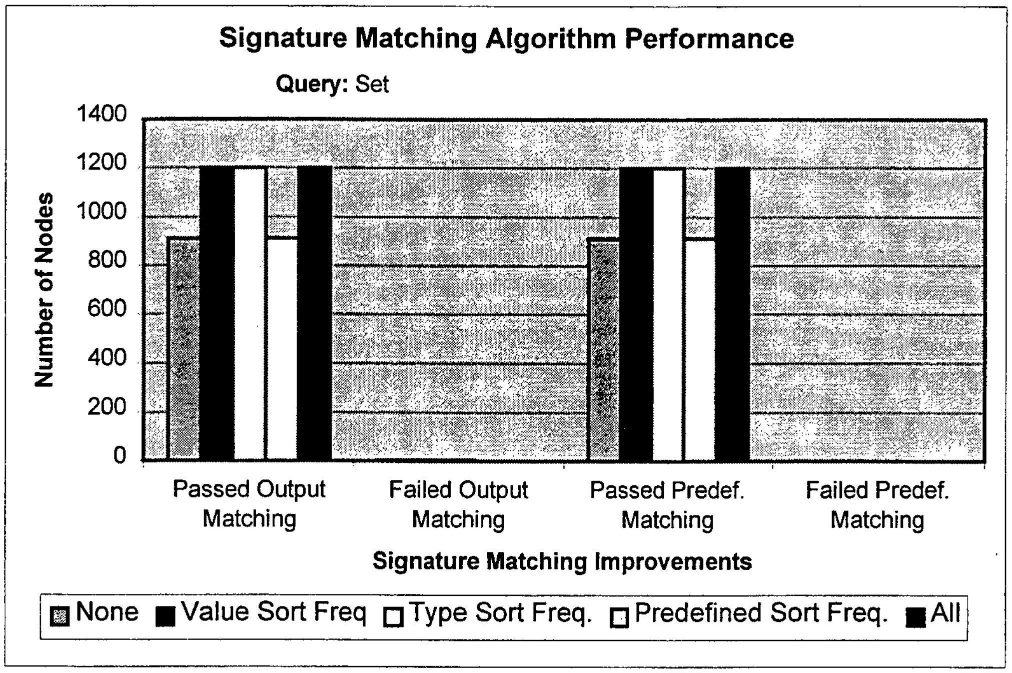


Figure 33: Signature Matching Algorithm Performance with Set Query

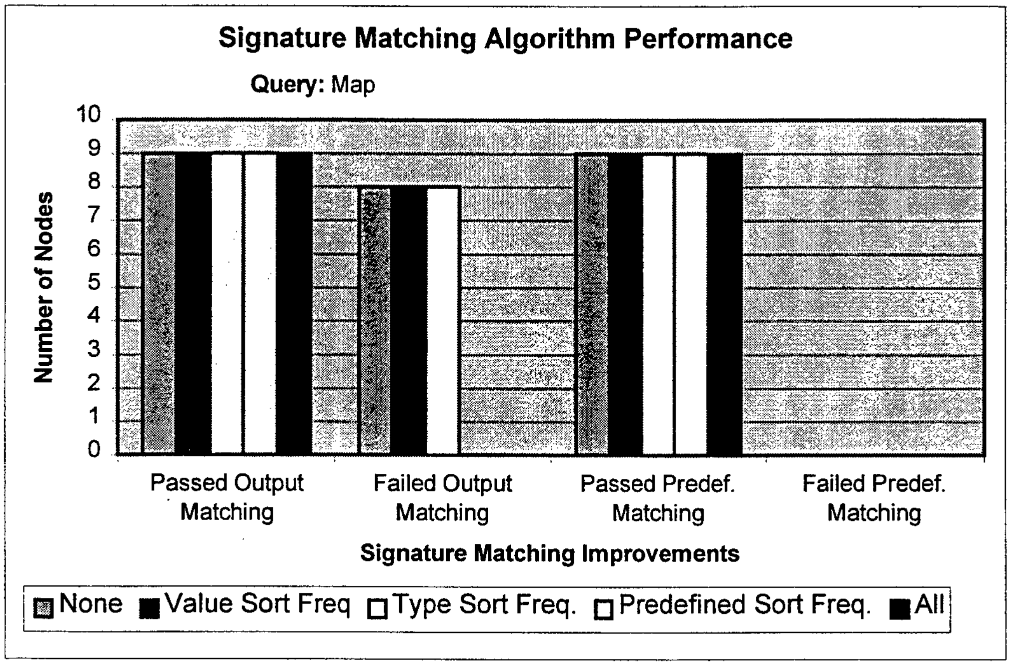


Figure 34: Signature Matching Algorithm Performance with Map Query

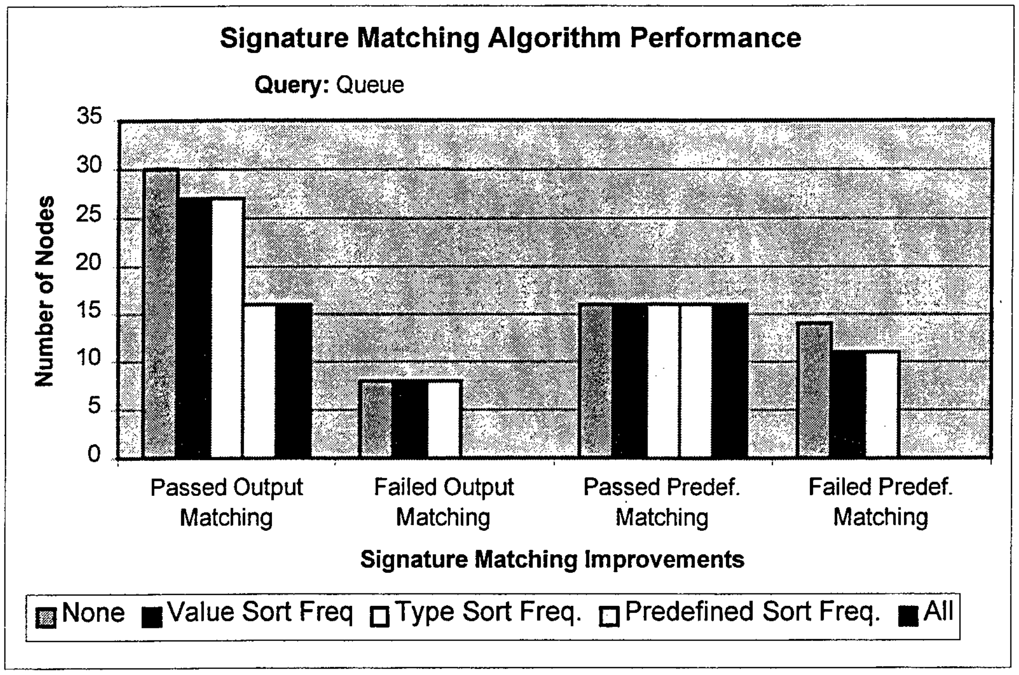


Figure 35: Signature Matching Algorithm Performance with Queue Query

Observing the number of failed nodes, however, does not consider the notion that the profile resolution improvements implicitly "prune" during the profile matching process. For example, the predefined sort frequency profile improvement, in many cases, can beat the predefined-matching signature matching improvement to the punch because it causes less operation pairs to be generated for operations with predefined types (operation pairs are generated when an operation in the query has an equal profile with an operation in the candidate). Hence, to merely look for a large number of failed nodes does not properly measure the full effectiveness of the complete syntactic filtering improvements outlined in this thesis because of the lack of orthogonality between profile filtering and signature matching.

VII. CONCLUSION AND FUTURE RESEARCH

A. ACCOMPLISHMENTS

This thesis has presented improvements to profile filtering and signature matching that help multi-level filtering achieve its goal of reducing large amounts of candidate components early in the process. More specifically, the resolution improvements to syntactic profiles enable the profile filtering process to significantly cut down the number of components passed on to the more computationally intensive signature matching process. Furthermore, we have seen that large-integer representations of syntactic profiles and exclusive use of a profile lookup table can lead to an optimal time-and-space implementation.

The improvements to signature matching included techniques for pruning the search-space of signature maps in an effort to find valid mappings quicker and with less computational resources. Initial experiments have backed up the theoretical instinct that the signature matching improvements are sensitive to the profile resolution improvements.

Finally, a detailed design and implementation of a syntactic matching software module that includes the improvements proposed in this thesis has been developed. The software has been written in Ada 95 and is mature enough for future inclusion with the other elements of multi-level filtering and CASE tools such as CAPS.

B. FUTURE RESEARCH

Future research should include more experiments with different software bases to better measure the effectiveness of the profile resolution improvements. Additionally, more data could be collected to better assess the effect profile resolution improvements have on signature matching.

The implementation facilitated the collection of statistics for generic component expansion. More software bases with generic components should be experimented with to gain more insight into the bloat generic components so quickly create. Further research into generic queries would also be insightful. The algorithms that instantiated generic components for search and retrieval preparation can also be used to instantiate a generic query. A study in using concurrent search and retrieval processes for each instantiation would certainly prove interesting.

Finally, research into effective graphical user interfaces for the user is needed. The multi-level filtering concept is natural for supporting incremental updates of query results, much like a web-browser incrementally updates information from a web-page. As the efficient front-end filters finish they provide early results that can be output to the user quickly. The user can then either select from these results or let the search process continue refining them. Either way, the user is given quick feedback that is important for user acceptance.

REFERENCES

[1] Valdis Berzins, CS4570 Class Notes, Naval Postgraduate School, Fall 1996.

[2] Scott Dolgoff, "Automated Interface for Retrieving Reusable Software Components", Master's Thesis, Naval Postgraduate School, September 1992.

[3] Joseph Goguen, Timothy Winkler, "Introducing OBJ3", Computer Science Laboratory, SRI International, SRI-CSL-88-9, August 1988.

[4] Himsolt, "GML: Graph Modeling Language", Draft, Deutsche Forschungsgemeinschaft Grant Br 835/6-2, December 1996.

[5] Luqi, "A Prototyping Language for Real-time Software", *IEEE Transactions of Software Engineering*, Vol. 14, No. 10, pp. 1409-1423, October 1988.

[6] Luqi, "Normalized Specifications for Identifying Reusable Software", *Proceedings of the 1987 Fall Joint Computer Conference*, pp. 46-49, IEEE, October, 1987.

[7] Luqi, and M. Ketabchi, "A Computer-Aided Prototyping System," *IEEE Transactions on Software Engineering*, October 1988.

[8] Mili, R. Mili, R. Mittermeir, "Storing and Retrieving Software Components", *Proceedings 16th International Conference on Software Engineering*, pp. 15-19, 1994.

[9] Doan Nguyen, "An Architectural Model for Software Component Search", Ph.D. Dissertation, Naval Postgraduate School, December 1995.

[10] Tuan Nguyen, "Populating the Software Database", Master's Thesis, Naval Postgraduate School, March 1996.

[11] Rubin Prieto-Diaz, "Implementing Faceted Classification for Software Reuse", *Communication of the ACM*, pp. 89-97, May 1991.

[12] Grady Booch, Ivar Jacobson, James Rumbaugh, UML Documentation Set, Rational Software Corporation, January 1997.

[13] Robert Steigerwald, Luqi, and John McDowell, "CASE Tool for Reusable Software Component Storage and Retrieval in Rapid Prototyping.", *Information and Software Technology*, pp. 698-705, 1991.

APPENDIX – SOURCE CODE

Makefile

```
#PSDL_TYPE_ROOT = /home/jsherman/MSSE/PSDL_TYPE-May97
PSDL_TYPE_ROOT = /home2/jsherman/PSDL_TYPE-May97

GEN = m4 generator.m4
#GEN = gen

INCLUDES = -I$(PSDL_TYPE_ROOT)/GNAT -I$(PSDL_TYPE_ROOT)/GENERIC_TYPES/GNAT -
I$(PSDL_TYPE_ROOT)/INSTANTIATIONS/GNAT

GEN_OBJECTS = candidate_types.adb haase_diagram.adb profile_calc.adb
profile_filter_pkg.adb psdl_profile.adb run_batch.adb sig_match.adb sig_match_types.adb
software_base.adb

.SUFFIXES: .g .adb

.g.adb:
	$(GEN) $< > $@

#all: run_batch test_profile_calc
all: run_batch

run_batch: $(GEN_OBJECTS)
	gnatmake $(INCLUDES) run_batch.adb

test_profile_calc: $(GEN_OBJECTS)
	gnatmake $(INCLUDES) test_profile_calc.adb

clean:
	rm -f *.o *.ali $(GEN_OBJECTS) test_profile_calc run_batch

cleangen:
	rm -f $(GEN_OBJECTS)
```

candidate_types.ads

```
------------------------------------------------------------------------
-- Package Spec: candidate_types
------------------------------------------------------------------------

with generic_sequence_pkg;
with ordered_set_pkg;
with component_id_types; use component_id_types;
with sig_match_types; use sig_match_types;

package candidate_types is

 RANK_UNKNOWN: constant := -1.0;

 --
 -- Candidate
 --
 type Candidate is record
    profile_rank: float;
    keyword_rank: float;
    signature_matches: SigMatchNodePtrSet;
    component_id: ComponentID;
 end record;

 function candidateEqual(c1: in Candidate; c2: in Candidate) return boolean;
 function candidateLessThan(c1: in Candidate; c2: in Candidate) return boolean;
 procedure candidateAssign(c1: in out Candidate; c2: in Candidate);
 procedure candidatePut(the_candidate: in Candidate);
 procedure candidatePrint(the_candidate: in Candidate);

 function newCandidate return Candidate;
 procedure generateSigMatchHistogram(filename: in string; c: in Candidate);

 --
 -- CandidateSequence
 --
 -- Note: should use addCandidate to add a candidate to the CandidateSequence.
 --       addCandidate keeps the CandidateSequence sorted.
 --
 package candidate_sequence_pkg is new generic_sequence_pkg(
    t => Candidate, average_size => 4);
 subtype CandidateSequence is candidate_sequence_pkg.sequence;

 function candidateSequenceEqual is
    new candidate_sequence_pkg.generic_equal(eq => candidateEqual);

 function candidateSequenceMember is
    new candidate_sequence_pkg.generic_member(eq => candidateEqual);

 procedure candidateSequenceRemove is
    new candidate_sequence_pkg.generic_remove(eq => candidateEqual);

 function candidateSequenceSort is
    new candidate_sequence_pkg.generic_sort("<" => candidateLessThan);

 procedure candidateSequencePut is
    new candidate_sequence_pkg.generic_put(put => candidatePut);

 procedure addCandidate(c: in Candidate; cs: in out CandidateSequence);

 --
 -- CandidateSet
 --
 package candidate_set_pkg is new ordered_set_pkg(t => Candidate,
    eq => candidateEqual, "<" => candidateLessThan);
 subtype CandidateSet is candidate_set_pkg.set;

 procedure candidateSetPut is
    new candidate_set_pkg.generic_put(put => candidatePut);

 function profileSkim(profile_threshold: in float;
    the_candidates: in CandidateSet) return CandidateSet;

 procedure generateProfileHistogram(filename: in string;
    the_candidates: in CandidateSet);
end candidate_types;
```

candidate_types.g

```
----------------------------------------------------------------------
-- Package Body: candidate_types
----------------------------------------------------------------------

with gnat.io;
with ada.text_io;
with ada.float_text_io;
with ada.integer_text_io;

with component_id_types; use component_id_types;

package body candidate_types is

 --
 -- Function: candidateEqual
 --
 function candidateEqual(c1: in Candidate; c2: in Candidate) return boolean is
 begin
    return c1.component_id = c2.component_id;
 end candidateEqual;

 --
 -- Function: candidateLessThan
 --
 -- Description: sort candidates in rank descending order (highest
 --              rank first).
 --
 function candidateLessThan(c1: in Candidate; c2: in Candidate) return boolean is
 begin
    -- TODO
    if c1.profile_rank > c2.profile_rank then
       return true;
    -- the followin test for less-than is just being paranoid
    -- about potential float equality problems
    elsif c1.profile_rank < c2.profile_rank then
       return false;
    else
       return c1.component_id < c2.component_id;
    end if;
 end candidateLessThan;

 --
 -- Procedure: candidateAssign
 --
 -- Description: makes a safe copy of a Candidate.  This is primarily
 --              necessary because of the SigMatchNodeSet
 --
 procedure candidateAssign(c1: in out Candidate; c2: in Candidate) is
 begin
    c1.profile_rank := c2.profile_rank;
    c1.keyword_rank := c2.keyword_rank;
    c1.component_id := c2.component_id;
    sig_match_node_ptr_set_pkg.assign(c1.signature_matches,
       c2.signature_matches);
 end candidateAssign;

 --
 -- Procedure: candidatePut
 --
 procedure candidatePut(the_candidate: in Candidate) is
 begin
    gnat.io.put("(");
    gnat.io.put(the_candidate.component_id);
    gnat.io.put(" | ");
    ada.float_text_io.put(the_candidate.profile_rank, 1, 2, 0);
    gnat.io.put(" | ");
    sigMatchNodePtrSetPut(the_candidate.signature_matches);
    gnat.io.put(")");
 end candidatePut;

 --
 -- Procedure: candidatePrint
 --
```

```
procedure candidatePrint(the_candidate: in Candidate) is
begin
   gnat.io.put("Component ID: ");
   gnat.io.put(the_candidate.component_id);
   gnat.io.new_line;
   gnat.io.put("Profile Rank: ");
   ada.float_text_io.put(the_candidate.profile_rank, 1, 2, 0);
   gnat.io.new_line;
   gnat.io.put(sig_match_node_ptr_set_pkg.size(
       the_candidate.signature_matches));
   gnat.io.put(" Signature Match Solutions:");
   gnat.io.new_line;
   sigMatchNodePtrSetPrint(the_candidate.signature_matches);
end candidatePrint;

--
-- Function: newCandidate
--
function newCandidate return Candidate is
   return_val: Candidate;
begin
   return_val.profile_rank := RANK_UNKNOWN;
   return_val.keyword_rank := RANK_UNKNOWN;
   return_val.signature_matches := sig_match_node_ptr_set_pkg.empty;
   return return_val;
end newCandidate;

--
-- generateSigMatchHistogram
--
-- Description: generates histogram data of the signature ranks for the
--              set of signature matches and saves it to a file so it can be
--              read by a charting program.  The format is one line
--              for each pair where the first item of the pair is the
--              profile rank and the second item is the number of
--              candidates with that rank.
--
procedure generateSigMatchHistogram(filename: in string; c: in Candidate) is
   ft: ada.text_io.file_type;
   last_rank: float;
   count: natural := 0;
   temp_snp: SigMatchNodePtr;

   procedure putPair(the_rank: float; the_count: natural) is
   begin
       ada.float_text_io.put(ft, the_rank, 1, 2, 0);
       ada.text_io.put(ft, " ");
       ada.integer_text_io.put(ft, the_count);
       ada.text_io.new_line(ft);
   end putPair;

begin
   ada.text_io.create(ft, ada.text_io.out_file, filename);

   if sig_match_node_ptr_set_pkg.size(c.signature_matches) = 0 then
       ada.text_io.close(ft);
       return;
   end if;

   temp_snp := sig_match_node_ptr_set_pkg.fetch(c.signature_matches, 1);
   last_rank := temp_snp.signature_rank;
   foreach((snp: SigMatchNodePtr), sig_match_node_ptr_set_pkg.scan,
           (c.signature_matches),
       if snp.signature_rank /= last_rank then
           putPair(last_rank, count);
           last_rank := snp.signature_rank;
           count := 1;
       else
           count := count + 1;
       end if;
   )
   putPair(last_rank, count);

   ada.text_io.close(ft);
end generateSigMatchHistogram;
```

```
--
-- Procedure: addCandidate
--
procedure addCandidate(c: in Candidate; cs: in out CandidateSequence) is
begin
   candidate_sequence_pkg.add(c, cs);
   cs := candidateSequenceSort(cs);
end addCandidate;

--
-- Function: profileSkim (for CandidateSet)
--
-- Description: filters out the candidates that do not meet the given
--              profile threshold.
--
function profileSkim(profile_threshold: in float;
       the_candidates: in CandidateSet) return CandidateSet is
   return_val: CandidateSet;
begin
   return_val := candidate_set_pkg.empty;
   foreach((c: Candidate), candidate_set_pkg.scan, (the_candidates),
       if c.profile_rank >= profile_threshold then
           candidate_set_pkg.add(c, return_val);
       end if;
   )
   return return_val;
end profileSkim;

--
-- Procedure: generateProfileHistogram
--
-- Description: generates histogram data of the profile ranks for the
--              set of candidates and saves it to a file so it can be
--              read by a charting program.  The format is one line
--              for each pair where the first item of the pair is the
--              profile rank and the second item is the number of
--              candidates with that rank.
--
procedure generateProfileHistogram(filename: in string;
       the_candidates: CandidateSet) is
   ft: ada.text_io.file_type;
   last_rank: float;
   count: natural := 0;
   temp_candidate: Candidate;

   procedure putPair(the_rank: float; the_count: natural) is
   begin
       ada.float_text_io.put(ft, the_rank, 1, 2, 0);
       ada.text_io.put(ft, " ");
       ada.integer_text_io.put(ft, the_count);
       ada.text_io.new_line(ft);
   end putPair;

begin
   ada.text_io.create(ft, ada.text_io.out_file, filename);

   if candidate_set_pkg.size(the_candidates) = 0 then
       ada.text_io.close(ft);
       return;
   end if;

   temp_candidate := candidate_set_pkg.fetch(the_candidates, 1);
   last_rank := temp_candidate.profile_rank;
   foreach((c: Candidate), candidate_set_pkg.scan, (the_candidates),
       if c.profile_rank /= last_rank then
           putPair(last_rank, count);
           last_rank := c.profile_rank;
           count := 1;
       else
           count := count + 1;
       end if;
   )
   putPair(last_rank, count);

   ada.text_io.close(ft);
end generateProfileHistogram;
end candidate_types;
```

component_id_types.ads

```
----------------------------------------------------------------------
-- Package Spec: component_id_types
----------------------------------------------------------------------

with gnat.io;

with generic_map_pkg;
with generic_set_pkg;
with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;

with psdl_profile; use psdl_profile;

package component_id_types is

 --
 -- ComponentID
 --
 subtype ComponentID is integer;

 procedure componentIDPut(c_id: ComponentID);

 --
 -- Component
 --
 -- Note: Make sure to use createComponent to instantiate a new Component.
 --       This will ensure that generics_mapping is initialized.
 --
 type Component is record
    psdl_filename: text;
    generics_mapping: GenericsMap;
 end record;

 function createComponent return Component;

 procedure addGenericsMapping(generic_type_id: psdl_id;
    actual_type_id: psdl_id; the_component:in out Component);

 function componentEqual(c1: in Component; c2: in Component) return boolean;

 procedure componentPut(the_component: in Component);

 --
 -- ComponentIDMap
 --
 package component_id_map_pkg is new generic_map_pkg(
    key => ComponentID,
    result => Component,
    eq_key => "=",
    eq_res => ComponentEqual,
    average_size => 8);
 subtype ComponentIDMap is component_id_map_pkg.map;

 procedure componentIDMapPut is new component_id_map_pkg.generic_put(
    key_put => gnat.io.put, res_put => componentPut);

 --
 -- ComponentIDSet
 --
 package component_id_set_pkg is new generic_set_pkg(
    t => ComponentID,
    average_size => 8,
    eq => "=");
 subtype ComponentIDSet is component_id_set_pkg.set;

 procedure componentIDSetPut is
    new component_id_set_pkg.generic_put(put => gnat.io.put);

 procedure componentIDSetFilePut is
    new component_id_set_pkg.generic_file_put(put => componentIDPut);

end component_id_types;
```

component_id_types.adb

```
-----------------------------------------------------------------------
-- Package Body: component_id_types
-----------------------------------------------------------------------
with gnat.io;
with text_io;

with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;

package body component_id_types is

 --
 -- Procedure: componentIDPut
 --
 procedure componentIDPut(c_id: ComponentID) is
 begin
    text_io.put(integer'image(c_id));
 end componentIDPut;

 --
 -- Procedure: createComponent
 --
 function createComponent return Component is
    return_val: Component;
 begin
    generics_map_pkg.create(empty, return_val.generics_mapping);
    return return_val;
 end createComponent;

 --
 -- Procedure: addGenericsMapping
 --
 procedure addGenericsMapping(generic_type_id: psdl_id;
    actual_type_id: psdl_id; the_component: in out Component) is
 begin
    generics_map_pkg.bind(generic_type_id, actual_type_id,
        the_component.generics_mapping);
 end addGenericsMapping;

 --
 -- Function: componentEqual
 --
 function componentEqual(c1: in Component; c2: in Component) return boolean is
 begin
    if not eq(c1.psdl_filename, c2.psdl_filename) then
        return false;
    end if;

    return generics_map_pkg.equal(c1.generics_mapping, c2.generics_mapping);
 end componentEqual;

 --
 -- Procedure: componentPut
 --
 procedure componentPut(the_component: in Component) is
 begin
    gnat.io.put(convert(the_component.psdl_filename));
    gnat.io.put(" | ");
    genericsMapPut(the_component.generics_mapping);
 end componentPut;

end component_id_types;
```

haase_diagram.ads

```
------------------------------------------------------------------------
-- Package Spec: haase_diagram
------------------------------------------------------------------------

with generic_map_pkg;

with profile_types; use profile_types;
with component_id_types; use component_id_types;

package haase_diagram is

 --
 -- Types
 --
 -- type HaaseNode is private;
 -- type HaaseDiagram is private;

 --
 -- HaaseNode
 --
 type HaaseNode is record
    key: ComponentProfile;
    components: ComponentIDSet;
    children: ComponentProfileSet;
 end record;

 function haaseNodeEqual(hn1: in HaaseNode; hn2: in HaaseNode)
    return boolean;

 procedure haaseNodeAssign(hn1: in out HaaseNode; hn2: in HaaseNode);

 procedure haaseNodePut(the_haase_node: in HaaseNode);

 procedure haaseNodePrint(the_haase_node: HaaseNode);

 --
 -- HaaseDiagram
 --
 package haase_node_map_pkg is new generic_map_pkg(
    key => ComponentProfile,
    result => HaaseNode,
    eq_key => componentProfileEqual,
    eq_res => haaseNodeEqual,
    average_size => 8);
 subtype HaaseDiagram is haase_node_map_pkg.map;

 procedure haaseDiagramPut is new haase_node_map_pkg.generic_put(
    key_put => componentProfilePut, res_put => haaseNodePut);

 procedure haaseDiagramPrint(the_haase_diagram: HaaseDiagram);

 procedure generateGML(the_haase_diagram: in HaaseDiagram;
    filename: in string);

 --
 -- Operations
 --
 function createHaaseNode(key: in ComponentProfile) return HaaseNode;
 function createHaaseDiagram return HaaseDiagram;

 procedure addComponent(the_comp_id: in ComponentID;
    the_haase_node: in out HaaseNode);

 procedure addChild(the_child_key: in ComponentProfile;
    the_haase_node: in out HaaseNode);

 procedure addHaaseNode(the_haase_node: in HaaseNode;
    the_haase_diagram: in out HaaseDiagram);

 procedure addBaseNodes(the_haase_diagram: in out HaaseDiagram);

 procedure connectNodes(the_haase_diagram: in out HaaseDiagram);
```

```
------------------------------------------------------------------------

-- private

end haase_diagram;
```

haase_diagram.g

```
-----------------------------------------------------------------------
-- Package Body: haase_diagram
-----------------------------------------------------------------------

with text_io; use text_io;

with generic_map_pkg;

with profile_types; use profile_types;
with component_id_types; use component_id_types;
with psdl_profile; use psdl_profile;
with software_base;

package body haase_diagram is

    --
    -- Function: createHaaseNode
    --
    -- Description: create and initialize a HaaseNode for use.
    --
    function createHaaseNode(key: in ComponentProfile) return HaaseNode is
        return_val: HaaseNode;
    begin
        profile_id_sequence_pkg.assign(return_val.key, key);
        return_val.components := component_id_set_pkg.empty;
        return_val.children := component_profile_set_pkg.empty;
        return return_val;
    end createHaaseNode;

    --
    -- Function: createHaaseDiagram
    --
    -- Description: create and initialize a HaaseDiagram for use.
    --
    function createHaaseDiagram return HaaseDiagram is
    begin
        return haase_node_map_pkg.create(
            createHaaseNode(profile_id_sequence_pkg.empty));
    end createHaaseDiagram;

    --
    -- Function: addComponent
    --
    -- Description: add a ComponentID to the HaaseNode.
    --
    procedure addComponent(the_comp_id: in ComponentID;
        the_haase_node: in out HaaseNode) is
    begin
        component_id_set_pkg.add(the_comp_id, the_haase_node.components);
    end addComponent;

    --
    -- Function: addChild
    --
    -- Description: add a ComponentProfile that represents the
    --              key to a child HaaseNode to the HaaseNode.
    --
    procedure addChild(the_child_key: in ComponentProfile;
        the_haase_node: in out HaaseNode) is
    begin
        component_profile_set_pkg.add(the_child_key, the_haase_node.children);
    end addChild;

    --
    -- Function: addHaaseNode
    --
    -- Description: add a HaaseNode to the HaaseDiagram.
    --
    procedure addHaaseNode(the_haase_node: in HaaseNode;
        the_haase_diagram: in out HaaseDiagram) is
        temp_key: ComponentProfile;
    begin
        profile_id_sequence_pkg.assign(temp_key, the_haase_node.key);
    haase_node_map_pkg.bind(temp_key, the_haase_node, the_haase_diagram);
end addHaaseNode;

--
-- Procedure: addBaseNodes
--
-- Description: add base nodes for the nodes already in the diagram.
--              This is done by adding a node for each profile in
--              the key for each node in the diagram.  Note, duplicates
--              will not be added.
--
procedure addBaseNodes(the_haase_diagram: in out HaaseDiagram) is
    new_diagram: HaaseDiagram;
    new_node: HaaseNode;
    new_key: ComponentProfile;
begin
    new_diagram := createHaaseDiagram;
    haase_node_map_pkg.assign(new_diagram, the_haase_diagram);
    new_key := profile_id_sequence_pkg.empty;

    -- for each((node_key: ComponentProfile; node: HaaseNode),
    --      haase_node_map_pkg.scan, (the_haase_diagram),

    --  for each((p_id: ProfileID), profile_id_sequence_pkg.scan,
    --          (node_key),
        foreach((p_id: ProfileID),
                profile_lookup_table_pkg.res_set_pkg.scan,
                (software_base.getProfileIDs),
            addProfileID(p_id, new_key);
            if not haase_node_map_pkg.member(new_key, the_haase_diagram) then
                new_node := createHaaseNode(new_key);
                addHaaseNode(new_node, new_diagram);
            end if;
            new_key := profile_id_sequence_pkg.empty;
        )
    -- )

    haase_node_map_pkg.assign(the_haase_diagram, new_diagram);
    haase_node_map_pkg.recycle(new_diagram);
end addBaseNodes;

--
-- Procedure: connectNodes
--
-- Description: connect nodes in diagram.  Invariant:
--              n2 is n1's child iff subbag(n1.key, n2.key) and
--              there is no node n3 such that subbag(n1.key, n3.key)
--              and subbag(n3.key, n2.key).
--
--             Note, an entirely new diagram is constructed because
--              scan returns copies of the nodes in the_haase_diagram,
--             not the actual nodes.
--
procedure connectNodes(the_haase_diagram: in out HaaseDiagram) is
    new_node: HaaseNode;
    new_diagram: HaaseDiagram;
    found_n3: boolean;
begin
    new_diagram := createHaaseDiagram;
    foreach((n1_key: ComponentProfile; n1: HaaseNode),
            haase_node_map_pkg.scan, (the_haase_diagram),
        new_node := createHaaseNode(n1_key);
        haaseNodeAssign(new_node, n1);

        foreach((n2_key: ComponentProfile; n2: HaaseNode),
                haase_node_map_pkg.scan, (the_haase_diagram),
            if not haaseNodeEqual(n1,n2) then
                if subbag(n1_key, n2_key) then
                    found_n3 := false;
                    foreach((n3_key: ComponentProfile; n3: HaaseNode),
                            haase_node_map_pkg.scan, (the_haase_diagram),
                        if not found_n3 then
                            if (not haaseNodeEqual(n1,n3)) and
                                         (not haaseNodeEqual(n2,n3)) then
                                if subbag(n1_key, n3_key) and
                                             subbag(n3_key, n2_key) then
```

```
                                                found_n3 := true;
                                          end if;
                                    end if;
                                 end if;
                              )
                              if not found_n3 then
                                 addChild(n2_key, new_node);
                              end if;
                           end if;
                        end if;
                     )
                     addHaaseNode(new_node, new_diagram);
                  )
                  haase_node_map_pkg.assign(the_haase_diagram, new_diagram);
                  haase_node_map_pkg.recycle(new_diagram);
               end connectNodes;

               --
               -- Function: haaseNodeEqual
               --
               -- Description: checks for equality of two haase nodes by
               --              comparing the keys.
               --
               function haaseNodeEqual(hn1: in HaaseNode; hn2: in HaaseNode)
                   return boolean is
               begin
                   return componentProfileEqual(hn1.key, hn2.key);
               end haaseNodeEqual;

               --
               -- Procedure: haaseNodeAssign
               --
               -- Description: creates a duplicate of hn2.
               --
               procedure haaseNodeAssign(hn1: in out HaaseNode; hn2: in HaaseNode) is
               begin
                   profile_id_sequence_pkg.assign(hn1.key, hn2.key);
                   component_id_set_pkg.assign(hn1.components, hn2.components);
                   -- component_profile_set_pkg.assign(hn1.children, hn2.children);
               end haaseNodeAssign;

               --
               -- Procedure: haaseNodePut
               --
               procedure haaseNodePut(the_haase_node: in HaaseNode) is
               begin
                   componentProfilePut(the_haase_node.key);
                   put("|");
                   componentIDSetPut(the_haase_node.components);
                   put("|");
                   componentProfileSetPut(the_haase_node.children);
               end haaseNodePut;

               --
               -- Procedure: haaseNodePrint
               --
               procedure haaseNodePrint(the_haase_node: in HaaseNode) is
               begin
                   put("Key: ");
                   componentProfilePut(the_haase_node.key);
                   new_line;
                   put("Components: ");
                   componentIDSetPut(the_haase_node.components);
                   new_line;
                   put("Children: ");
                   componentProfileSetPut(the_haase_node.children);
                   new_line;
               end haaseNodePrint;

               --
               -- Procedure: haaseDiagramPrint
               --
               procedure haaseDiagramPrint(the_haase_diagram: in HaaseDiagram) is
               begin
                   foreach((node_key: ComponentProfile; node: HaaseNode),
                           haase_node_map_pkg.scan, (the_haase_diagram),
        haaseNodePrint(node);
        new_line;
    )
    new_line;
end haaseDiagramPrint;

--
-- Procedure: generateGML
--
-- Description: generate a GML file to graphically represent the
--              HaaseDiagram.
--
procedure generateGML(the_haase_diagram: in HaaseDiagram;
        filename: in string) is
    id: natural := 0; -- unique ID counter
    the_id: natural;
    gml_file: file_type;

    function new_id return natural is
    begin
        id := id + 1;
        return id;
    end new_id;

    package temp_map_pkg is new generic_map_pkg(
        key => ComponentProfile,
        result => natural,
        eq_key => componentProfileEqual,
        eq_res => "=",
        average_size => 8);
    subtype tempMap is temp_map_pkg.map;

    temp_map: tempMap;

begin
    create(gml_file, out_file, filename);
    put(gml_file, "graph [ id ");
    put(gml_file, integer'image(new_id));
    put_line(gml_file, " directed 1");

    temp_map_pkg.create(id, temp_map);

    -- make the nodes
    foreach((node_key: ComponentProfile; node: HaaseNode),
            haase_node_map_pkg.scan, (the_haase_diagram),
        put(gml_file, "node [ id ");
        the_id := new_id;
        put(gml_file, integer'image(the_id));
        put(gml_file, " label """);
        componentProfileFilePut(gml_file, node.key);
        -- put_line(gml_file, "\");
        -- componentIDSetFilePut(gml_file, node.components);
        put_line(gml_file, """ ]");

        temp_map_pkg.bind(node.key, the_id, temp_map);
    )

    -- make the edges
    foreach((node_key: ComponentProfile; node: HaaseNode),
            haase_node_map_pkg.scan, (the_haase_diagram),
        foreach((child_key: ComponentProfile),
                component_profile_set_pkg.scan, (node.children),
            put(gml_file, "edge [ id ");
            put(gml_file, integer'image(new_id));
            put(gml_file, " source ");
            put(gml_file, integer'image(temp_map_pkg.fetch(temp_map,
                node.key)));
            put(gml_file, " target ");
            put(gml_file, integer'image(temp_map_pkg.fetch(temp_map,
                child_key)));
            put_line(gml_file, " ]");
        )
    )

    put_line(gml_file, "]");
    close(gml_file);
```

```
        temp_map_pkg.recycle(temp_map);
    end generateGML;

end haase_diagram;
```

profile_calc.ads

```
------------------------------------------------------------------------
-- Package Spec: profile_calc
--
-- This package contains functions and types that support the computation
-- of profiles from numeric representations of signatures.
--
-- Description of numeric signatures: Positive integers represent
-- instances of non-generic types in the signature.  Negative integers
-- represent instances of generic types in the signature.  Finally,
-- a 0 is used to terminate the array of integers representing the
-- signature.
--
-- Examples of numeric signatures:
-- [integer, char, float -> integer]  ==> [1,2,3,1,0]
-- [integer, generic, float -> float] ==> [1,-1,2,3,0]
-- [generic1, generic2 -> generic2]   ==> [-1,-2,-2,0]
--
-- Profiles are sequences of integers.
--
-- Generic Types:
-- Generic types cause more than one profile to be generated for a
-- single signature.  Hence, computeArrayProfileWithGenerics returns an
-- array of ArrayProfiles, ProfileValues, bound by NumProfiles.
--
-- ArrayProfiles are terminated with PROFILE_TERMINATOR.  For example,
-- the profile [3,1,1,2] is returned as [3,1,1,2,-99].
--
-- Eventually a different method for handling generic types will be
-- employed and will likely do away with the ArrayProfile data type.
------------------------------------------------------------------------

with profile_types; use profile_types;

package profile_calc is

 --
 -- Types
 --
 MAX_SIG_LENGTH: constant := 100;
 MAX_PROFILE_LENGTH: constant := 100;
 MAX_PROFILE_VARIATIONS: constant := 100; -- for generic types
 PROFILE_TERMINATOR: constant := -99;

 subtype SignatureLengthRange is Positive range 1..MAX_SIG_LENGTH;
 subtype ProfileLengthRange is Positive range 1..MAX_PROFILE_LENGTH;
 subtype ProfileVariationRange is Positive range 1..MAX_PROFILE_VARIATIONS;

 type Signature is array (SignatureLengthRange) of Integer;
 type ArrayProfile is array (ProfileLengthRange) of Integer;
 type ArrayProfiles is array (ProfileVariationRange) of ArrayProfile;

 --
 -- Functions
 --
 function computeProfile(T: in Signature) return Profile;
 function computeArrayProfile(T: in Signature) return ArrayProfile;

 -- note NumProfiles should be 0..MAX_PROFILE_VARIATIONS, not Natural
 procedure computeArrayProfileWithGenerics(
    T: in Signature;
    ProfileValues: out ArrayProfiles;
    NumProfiles: out Natural);

 function printSignature(sig: Signature) return SignatureLengthRange;
 function printArrayProfile(prof: ArrayProfile) return ProfileLengthRange;

end profile_calc;
```

profile_calc.g

```
-----------------------------------------------------------------
-- Package Body: profile_calc
-----------------------------------------------------------------
with gnat.io; use gnat.io;

with profile_types; use profile_types;

package body profile_calc is

  --
  -- Function: convertToSequence
  --
  -- Description: helper function to convert an ArrayProfile (an
  --              array of ints terminated with PROFILE_TERMINATOR)
  --              to a Profile (a sequence of ints).
  --
  function convertToSequence(Prof: ArrayProfile) return Profile is
    return_val: Profile;
    i, count: ProfileLengthRange;
  begin
    count := 1;
    while Prof(count) /= PROFILE_TERMINATOR and count <= MAX_PROFILE_LENGTH loop
        count := count + 1;
    end loop;
    count := count - 1;

    return_val := 0;
    for i in 1..count loop
        return_val := return_val + (long_long_integer(Prof(i)) *
            (10 ** (count-i)));
    end loop;

    return return_val;
  end convertToSequence;

  function printSignature(Sig: Signature) return SignatureLengthRange is
    Num: SignatureLengthRange;
  begin
    Num := 1;
    Put("[");
    while Sig(Num + 1) /= 0 loop
      Put (Sig(Num));
      if Sig(Num + 2) /= 0 then
        Put (", ");
      end if;
      Num := Num + 1;
    end loop;
    Put (" -> ");
    Put (Sig(Num));
    Put("]");
    return Num;
  end printSignature;

  function printArrayProfile(Prof: ArrayProfile) return ProfileLengthRange is
    Num: ProfileLengthRange;
  begin
    Num := 1;
    Put("[");
    while Prof(Num) /= PROFILE_TERMINATOR and Num < MAX_PROFILE_LENGTH loop
      Put (Prof(Num));
      if Prof(Num + 1) /= PROFILE_TERMINATOR then
        Put (", ");
      end if;
      Num := Num + 1;
    end loop;
    Put("]");
    return Num;
  end printArrayProfile;

  function computeProfile(T: Signature) return Profile is
  begin
    return convertToSequence(computeArrayProfile(T));
  end computeProfile;
```

```
function computeArrayProfile(T: Signature) return ArrayProfile is
  Result: ArrayProfile;
  Result_Count : Integer;
  NumResSort: Integer;
  NumOneSorts: Integer;
  I,J: Integer;
  L: SignatureLengthRange;
  SortValues: array (SignatureLengthRange) of Integer;
  SortNums: array (SignatureLengthRange) of Integer;
  NumSorts: Integer;
  Found: Boolean;
begin
  -- Compute Profile[1], Total Number of Sorts.
  Result_Count := 1;
  J := 0;

 -- set L to number of elements in T
 -- note, this is the first number in the profile
 I := 1;
 while (T(I) /= 0 and I <= MAX_SIG_LENGTH) loop
   I := I + 1;
 end loop;
 L := I - 1;

  Result(Result_Count) :=  L;

  -- Compute Profile[2], Number of Times Result Sort in Signature.
  -- note, Nguyen's thesis just uses 0 or 1 to indicate if the
  -- result sort is used in the input arguments.  Representing
  -- the number of times the result sort is used is finer resolution,
  -- which should partition of the software base better.
  NumResSort := 0;
  for I in 1..L loop
    if T(I) = T(L) then
       NumResSort := NumResSort + 1;
    end if;
  end loop;
  Result_Count := Result_Count + 1;

  -- Herman
  -- Result(Result_Count) :=  NumResSort;

  -- Nguyen
  if NumResSort > 1 then
      Result(Result_Count) :=  1;
  else
      Result(Result_Count) :=  0;
  end if;

  -- Herman Improvement Profile[3]
  -- Add the number of occurrences of the type being defined by the
  -- component (if the component is a type).
  --Result_Count := Result_Count + 1;
  --Result(Result_Count) := T(L+2);

  -- Herman Improvement Profile[4..8]
  -- Add the number of occurrences of types in the basic sort groups
  Result_Count := Result_Count + 1;
  Result(Result_Count) := T(L+3);
  Result_Count := Result_Count + 1;
  Result(Result_Count) := T(L+4);
  --Result_Count := Result_Count + 1;
  --Result(Result_Count) := T(L+5);
  Result_Count := Result_Count + 1;
  Result(Result_Count) := T(L+6);
  Result_Count := Result_Count + 1;
  Result(Result_Count) := T(L+7);

  -- Generate Helper Arrays
  -- SortValues: an ordered SET of sort values
  --    e.g. if the signature input T was [1, 1, 2, 1, 0]
  --         SortValues would be [1, 2]
  -- NumSorts: the cardinality of the ordered set SortValues
  --    e.g. in the above example, NumSorts would be 2
  -- SortNums: the cardinality of each sort in SortValues
  --   e.g. in the above example, SortValues would be [3, 1]
  for I in 1..L loop
    SortNums(I) := 0;
  end loop;
  SortValues(1) := T(1);
  NumSorts := 1;
  SortNums(1) := 1;
  for I in 2..L loop
    Found := False;
    for J in 1..NumSorts loop
      if T(I) = SortValues(J) then
         SortNums(J) := SortNums(J) + 1;
         Found := True;
      end if;
    end loop;
    if not Found then
      NumSorts := NumSorts + 1;
      SortValues(NumSorts) := T(I);
      SortNums(NumSorts) := 1;
    end if;
  end loop;

  -- Becomes Profile[9]
  -- Compute Profile[3], Number of Sort Groups of Size One.
  NumOneSorts := 0;
  for I in 1..NumSorts loop
    if SortNums(I) = 1 then
      NumOneSorts := NumOneSorts + 1;
    end if;
  end loop;
  Result_Count := Result_Count + 1;
  Result(Result_Count) :=  NumOneSorts;

  -- Becomes Profile[10..N]
  -- Compute Profile[4..N], Sequence of Sizes of the Sort Groups that
  -- Have Size Greater than One.
  for I in 0..L-2 loop
    for J in 1..NumSorts loop
      if SortNums(J) = L-I then
         Result_Count := Result_Count + 1;
         Result(Result_Count) :=  L-I;
      end if;
    end loop;
  end loop;

  -- Terminate the ArrayProfile
  Result(Result_Count+1) := PROFILE_TERMINATOR;
  return Result;
end computeArrayProfile;

procedure computeArrayProfileWithGenerics(
  T: in Signature;
  ProfileValues: out ArrayProfiles;
  NumProfiles: out Natural) is
 I, G, J, K: Integer;
 L: SignatureLengthRange;
 NewSig: Signature;
 NumGenerics:  Integer;
 NumDiffGenerics: Integer;
 Found: Boolean;
 Valj: Integer;
 GenericPos: array (SignatureLengthRange) of Integer;
 ProfileVal: ArrayProfile;
begin
 NumGenerics := 0;
 NumProfiles := 0;
 Valj:=0;
 NumDiffGenerics := 0;
 G := 0;
 J := 0;
 K := 0;

 -- set L to number of elements in T
 I := 1;
 while (T(I) /= 0 and I <= MAX_SIG_LENGTH) loop
   I := I + 1;
```

```
    end loop;
    L := I - 1;

    for I in 1..L loop
      if T(I) < 0 then
        if T(I) < NumDiffGenerics then
           NumDiffGenerics := T(I);
        end if;
        NumGenerics := NumGenerics + 1;
        GenericPos(NumGenerics) := I;
      end if;
    end loop;
    NumDiffGenerics := -1 * NumDiffGenerics ;
    if NumGenerics = 0 then
      NumProfiles := 1;
      ProfileVal := computeArrayProfile(T);
      ProfileValues(1) := ProfileVal;
    else
     for G in 1..NumDiffGenerics loop
      for I in 1..L loop
        NewSig(I) := T(I);
      end loop;
      NewSig(L+1) := 0;
      for J in 1..L loop
          for I in 1..NumGenerics loop
            if T(GenericPos(I)) >= -1 * G then
              NewSig(GenericPos(I)) := T(J);
            end if;
          end loop;
          --
          -- These following lines are good for debugging.
          -- They print out all the combinations of signatures computed
          Valj:= printSignature(NewSig);
          New_Line;
          --
          ProfileVal := computeArrayProfile(NewSig);
          if NumProfiles = 0 then
            NumProfiles := 1;
            ProfileValues(1) := ProfileVal;
          else
            Found := False;
            for K in 1..NumProfiles loop
              if ProfileValues(K) = ProfileVal then
                Found := True;
              end if;
            end loop;
            if not Found then
              NumProfiles := NumProfiles + 1;
              ProfileValues(NumProfiles) := ProfileVal;
            end if;
          end if;
      end loop;
     end loop;
    end if;
  end computeArrayProfileWithGenerics;

end profile_calc;
```

profile_filter_pkg.ads

```
-----------------------------------------------------------------------
-- Package Spec: profile_filter
-----------------------------------------------------------------------

with haase_diagram; use haase_diagram;
with candidate_types; use candidate_types;
with profile_types; use profile_types;

package profile_filter_pkg is

 function findCandidates(query_profile: in ComponentProfile;
    the_haase_diagram: in HaaseDiagram) return CandidateSet;

end profile_filter_pkg;
```

profile_filter_pkg.g

```
------------------------------------------------------------------------
-- Package Body: profile_filter
------------------------------------------------------------------------

with haase_diagram; use haase_diagram;
with candidate_types; use candidate_types;
with component_id_types; use component_id_types;

package body profile_filter_pkg is

 --
 -- Function: findCandidates
 --
 -- Description: for each profile in query_profile start at the base-node
 --              that represents that profile and perform a depth-first
 --              search on the haase-diagram.  At each node calculate the
 --              profile rank, create a Candidate with that rank and the
 --              components in that node, and add it to return_val.
 --
 function findCandidates(query_profile: in ComponentProfile;
       the_haase_diagram: in HaaseDiagram) return CandidateSet is
   return_val: CandidateSet;
   base_node: HaaseNode;
   base_node_key: ComponentProfile;
   num_matches: natural;
   i, j: natural;

   procedure DFSFW(hn: in HaaseNode) is
       temp_candidate: Candidate;
   begin
       -- count the number of profiles in the node that
       -- are also in the query
       num_matches := 0;
       i := 1;
       j := 1;
       while i <= profile_id_sequence_pkg.length(query_profile) and
               j <= profile_id_sequence_pkg.length(hn.key) loop
           if profile_id_sequence_pkg.fetch(query_profile, i) =
                   profile_id_sequence_pkg.fetch(hn.key, j) then
               num_matches := num_matches + 1;
               i := i + 1;
               j := j + 1;
           elsif profileIDLessThan(profile_id_sequence_pkg.fetch(query_profile, i),
                   profile_id_sequence_pkg.fetch(hn.key, j)) then
               i := i + 1;
           else
               j := j + 1;
           end if;
       end loop;

       -- add the node's components to return val
       foreach((comp_id: ComponentID), component_id_set_pkg.scan,
               (hn.components),
           temp_candidate := newCandidate;
           temp_candidate.profile_rank :=
               float(num_matches) / float(profile_id_sequence_pkg.length(query_profile));
           temp_candidate.component_id := comp_id;
           candidate_set_pkg.add(temp_candidate, return_val);
       )

       -- recursively call DFSFW on each child
       foreach((child: ComponentProfile), component_profile_set_pkg.scan,
               (hn.children),
           DFSFW(haase_node_map_pkg.fetch(the_haase_diagram, child));
       )
   end DFSFW;

 begin
   return_val := candidate_set_pkg.empty;

   foreach((p_id: ProfileID), profile_id_sequence_pkg.scan, (query_profile),
       base_node_key := profile_id_sequence_pkg.empty;
       addProfileID(p_id, base_node_key);
```

```
        if haase_node_map_pkg.member(base_node_key, the_haase_diagram) then
            base_node :=
                haase_node_map_pkg.fetch(the_haase_diagram, base_node_key);
            DFSFW(base_node);
        end if;
    )

    return return_val;
 end findCandidates;

end profile_filter_pkg;
```

profile_types.ads

```
---------------------------------------------------------------------------
-- Package Spec: profile_types
---------------------------------------------------------------------------

with gnat.io;

with generic_sequence_pkg;
with generic_set_pkg;
with ordered_map_pkg;

package profile_types is

 procedure myIntPut(i: integer);

 --
 -- Profile
 --
 -- package int_sequence_pkg is new generic_sequence_pkg(
 -- t => integer, average_size => 4);
 -- subtype Profile is int_sequence_pkg.sequence;

 -- function profileEqual is new int_sequence_pkg.generic_equal(eq => "=");
 -- function profileLessThan is new int_sequence_pkg.generic_less_than("<" => "<");
 -- procedure profilePut is new int_sequence_pkg.generic_put(put => gnat.io.put);
 -- procedure profileFilePut is new int_sequence_pkg.generic_put(put => myIntPut);

 subtype Profile is long_long_integer;

 function profileEqual(p1, p2: Profile) return boolean;
 function profileLessThan(p1, p2: Profile) return boolean;
 procedure profilePut(p: Profile);
 procedure profileFilePut(p: Profile);

 --
 -- ProfileID
 --
 subtype ProfileID is integer;

 function profileIDLessThan(p1, p2: ProfileID) return boolean;
 procedure profileIDPut(p_id: ProfileID);
 procedure profileIDFilePut(p_id: ProfileID);

 --
 -- ProfileLookupTable
 --
 DEFAULT_PROFILE_ID: constant := -1;
 package profile_lookup_table_pkg is new ordered_map_pkg(
    key => Profile,
    result => ProfileID,
    eq_key => profileEqual,
    eq_res => "=",
    "<" => profileLessThan);
 subtype ProfileLookupTable is profile_lookup_table_pkg.map;

 procedure profileLookupTablePut is new profile_lookup_table_pkg.generic_put(
    key_put => profilePut, res_put => profileIDPut);

 --
 -- ComponentProfile
 --
 -- Note: should use addProfileID to add a profile id to the ComponentProfile.
 --       addProfileID keeps the ComponentProfile sorted which is important
 --       for equality and subbag (multiset subset) testing.
 --
 package profile_id_sequence_pkg is new generic_sequence_pkg(
    t => ProfileID, average_size => 4);
 subtype ComponentProfile is profile_id_sequence_pkg.sequence;

 function componentProfileEqual is
    new profile_id_sequence_pkg.generic_equal(eq => "=");

 function componentProfileMember is
    new profile_id_sequence_pkg.generic_member(eq => "=");
 procedure componentProfileRemove is
    new profile_id_sequence_pkg.generic_remove(eq => "=");

 function componentProfileSort is
    new profile_id_sequence_pkg.generic_sort("<" => "<");

 function componentProfileLessThan is
    new profile_id_sequence_pkg.generic_less_than("<" => profileIDLessThan);

 procedure componentProfilePut is
    new profile_id_sequence_pkg.generic_put(put => profileIDPut);

 procedure componentProfileFilePut is
    new profile_id_sequence_pkg.generic_file_put(put => profileIDFilePut);

 function subbag is
    new profile_id_sequence_pkg.generic_subsequence(eq => "=");

 package component_profile_set_pkg is new generic_set_pkg(
    t => ComponentProfile, eq => componentProfileEqual, average_size => 8);
 subtype ComponentProfileSet is component_profile_set_pkg.set;

 procedure componentProfileSetPut is
    new component_profile_set_pkg.generic_put(put => componentProfilePut);

 procedure addProfileID(p_id: in ProfileID; cp: in out ComponentProfile);
 procedure addProfiles(new_profiles: in ComponentProfile;
    target: in out ComponentProfile);

end profile_types;
```

profile_types.adb

```
------------------------------------------------------------------------
-- Package Body: profile_types
------------------------------------------------------------------------

with text_io;
with ada.long_long_integer_text_io;
with software_base;

package body profile_types is

    --
    -- Procedure: myIntPut
    --
    procedure myIntPut(i: integer) is
    begin
        text_io.put(integer'image(i));
    end myIntPut;

    --
    -- Procedure: addProfileID
    --
    -- Description: adds a ProfileID to a ComponentProfile by adding the
    --              ProfileID to the sequence then sorting the sequence.
    --
    procedure addProfileID(p_id: in ProfileID; cp: in out ComponentProfile) is
    begin
        profile_id_sequence_pkg.add(p_id, cp);
        cp := componentProfileSort(cp);
    end addProfileID;

    --
    -- Procedure: addProfiles
    --
    -- Description: appends the profiles from new_profiles to target then
    --              sorts target.
    --
    procedure addProfiles(new_profiles: in ComponentProfile;
            target: in out ComponentProfile) is
    begin
        target := profile_id_sequence_pkg.append(target, new_profiles);
        target := componentProfileSort(target);
    end addProfiles;

    --
    -- Function: profileEqual
    --
    function profileEqual(p1, p2: Profile) return boolean is
    begin
        return p1 = p2;
    end profileEqual;

    --
    -- Function: profileLessThan
    --
    function profileLessThan(p1, p2: Profile) return boolean is
    begin
        return p1 < p2;
    end profileLessThan;

    --
    -- Function: profilePut
    --
    procedure profilePut(p: Profile) is
    begin
        ada.long_long_integer_text_io.put(p,0);
    end profilePut;

    --
    -- Function: profileFilePut
    --
    procedure profileFilePut(p: Profile) is
    begin
        profilePut(p);
    end profileFilePut;

    --
    -- Function: profileIDLessThan
    --
    function profileIDLessThan(p1, p2: ProfileID) return boolean is
    begin
        return software_base.getProfile(p1) < software_base.getProfile(p2);
    end profileIDLessThan;

    --
    -- Procedure: profileIDPut
    --
    procedure profileIDPut(p_id: ProfileID) is
    begin
        text_io.put(integer'image(p_id));
    end profileIDPut;

    --
    -- Function: profileIDFilePut
    --
    procedure profileIDFilePut(p_id: ProfileID) is
    begin
        profileIDPut(p_id);
    end profileIDFilePut;

    --
    -- Function: createProfileLookupTable
    --
    function createProfileLookupTable return ProfileLookupTable is
    begin
        return profile_lookup_table_pkg.create(0);
    end createProfileLookupTable;

end profile_types;
```

psdl_profile.ads

```
-----------------------------------------------------------------------
-- Package Spec: psdl_profile
--
-- This package contains functions and types that support the collection
-- of operation profiles from a component specified in PSDL.
-----------------------------------------------------------------------

with generic_sequence_pkg;
with generic_map_pkg;
with generic_set_pkg;
with ordered_set_pkg;

with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with psdl_component_pkg; use psdl_component_pkg;

with profile_types; use profile_types;

package psdl_profile is

 --
 -- Types
 --

 --
 -- OpWithProfile
 --
 type OpWithProfile is record
    op: operator;
    op_profile: ProfileID;
 end record;

 function opWithProfileEqual(owp1: in OpWithProfile; owp2: in OpWithProfile)
    return boolean;

 function opWithProfileLessThan(owp1: in OpWithProfile; owp2: in OpWithProfile)
    return boolean;

 procedure opWithProfilePut(owp: in OpWithProfile);

 --
 -- OpWithProfileSeq
 --
 -- Note: should use addOpWithProfile to add an OpWithProfile to the sequence.
 --       addOpWithProfile keeps the sequence sorted.
 --
 package owp_sequence_pkg is new generic_sequence_pkg(
    t => OpWithProfile, average_size => 4);
 subtype OpWithProfileSeq is owp_sequence_pkg.sequence;

 function opWithProfileSeqEqual is
    new owp_sequence_pkg.generic_equal(eq => opWithProfileEqual);

 function opWithProfileSeqMember is
    new owp_sequence_pkg.generic_member(eq => opWithProfileEqual);

 procedure opWithProfileSeqRemove is
    new owp_sequence_pkg.generic_remove(eq => opWithProfileEqual);

 function opWithProfileSeqSort is
    new owp_sequence_pkg.generic_sort("<" => opWithProfileLessThan);

 procedure opWithProfileSeqPut is
    new owp_sequence_pkg.generic_put(put => opWithProfilePut);

 procedure opWithProfileSeqPrint(owp_seq: in OpWithProfileSeq);

 procedure addOpWithProfile(owp: in OpWithProfile;
    owp_seq: in out OpWithProfileSeq);

 --
 -- OpWithProfileSet
 --
 package owp_set_pkg is new ordered_set_pkg(
     t => OpWithProfile, eq => opWithProfileEqual,
     "<" => opWithProfileLessThan);
  subtype OpWithProfileSet is owp_set_pkg.set;

  procedure opWithProfileSetPut is
     new owp_set_pkg.generic_put(put => opWithProfilePut);

  procedure opWithProfileSetPrint(owp_set: in OpWithProfileSet);

  --
  -- GenericsMap
  --
  -- Description: this is a mapping of generic type identifiers to
  -- actual types that exist in the component.  For example, if the
  -- PSDL type Stack has one generic type named Item and has methods
  -- that have parameters that use the types natural, Stack, and
  -- boolean then there would be four different instantiations of
  -- Stack in the software base representing the four possible
  -- mappings for Item:  1. Item => natural, 2. Item => Stack,
  -- 3. Item => boolean, 4. Item => Item.  Option 4 really just
  -- means that Item is mapped to a type that does not appear in the
  -- component.  Suppose Stack used two generic types.  In that case
  -- each instantiation's GenericsMap would have two entries, one
  -- for each generic type.  In such a case the number of different
  -- instantiations present in the software base grows rapidly;
  -- specifically the number would be the cross product of the number
  -- of types across each generic type.
  --
  package generics_map_pkg is new generic_map_pkg(
     key => psdl_id,
     result => psdl_id,
     eq_key => eq,
     eq_res => eq,
     average_size => 8);
  subtype GenericsMap is generics_map_pkg.map;

  procedure psdl_idPut(the_id: in psdl_id);

  procedure genericsMapPut is new generics_map_pkg.generic_put(
     key_put => psdl_idPut, res_put => psdl_idPut);

  --
  -- GenericsMapSet
  --
  package generics_map_set_pkg is new generic_set_pkg(
     t => GenericsMap, eq => generics_map_pkg.equal);
  subtype GenericsMapSet is generics_map_set_pkg.set;

  procedure genericsMapSetPut is
     new generics_map_set_pkg.generic_put(put => genericsMapPut);

  --
  -- Functions
  --
  function getGenericsMaps(filename: in string) return GenericsMapSet;

  function getComponentProfile(filename: in string;
     generics_mapping: in GenericsMap) return ComponentProfile;

  function getOpsWithProfiles(filename: in string;
     generics_mapping: in GenericsMap) return OpWithProfileSeq;

  function getOpsWithProfiles(filename: in string;
     generics_mapping: in GenericsMap) return OpWithProfileSet;

end psdl_profile;
```

psdl_profile.g

```
-----------------------------------------------------------------
-- Package Body: psdl_profile
-----------------------------------------------------------------
with text_io; use text_io;

with profile_types; use profile_types;
with profile_calc; use profile_calc;

with psdl_io;
with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with psdl_component_pkg; use psdl_component_pkg;
with psdl_program_pkg; use psdl_program_pkg;
with psdl_id_set_subtype_pkg;
with psdl_id_pkg;
with software_base;

with generic_map_pkg;
with generic_sequence_pkg;

package body psdl_profile is

    package signature_seq_pkg is new generic_sequence_pkg(
        t => Signature, average_size => 2);
    subtype SignatureSequence is signature_seq_pkg.sequence;

    --
    -- Function: opWithProfileEqual
    --
    function opWithProfileEqual(owp1: in OpWithProfile; owp2: in OpWithProfile)
        return boolean is
    begin
        -- if not profileEqual(owp1.op_profile, owp2.op_profile) then
        if owp1.op_profile /= owp2.op_profile then
            return false;
        end if;
        return eq(owp1.op, owp2.op);
    end opWithProfileEqual;

    --
    -- Function: opWithProfileLessThan
    --
    function opWithProfileLessThan(owp1: in OpWithProfile;
            owp2: in OpWithProfile)
        return boolean is
    begin
        -- return profileLessThan(owp1.op_profile, owp2.op_profile);
        return profileLessThan(software_base.getProfile(owp1.op_profile),
            software_base.getProfile(owp2.op_profile));
    end opWithProfileLessThan;

    --
    -- Function: opWithProfilePut
    --
    procedure opWithProfilePut(owp: in OpWithProfile) is
    begin
        put("(");
        put(convert(name(owp.op)));
        put(": ");
        foreach((the_id: psdl_id; the_tn: type_name),
                type_declaration_pkg.scan, (inputs(owp.op)),
            put(convert(the_tn.name));
            put(" ");
        )
        put("-> ");
        foreach((the_id: psdl_id; the_tn: type_name),
                type_declaration_pkg.scan, (outputs(owp.op)),
            put(convert(the_tn.name));
            put(" ");
        )
        put("| ");
        profilePut(software_base.getProfile(owp.op_profile));
        put(")");
    end opWithProfilePut;
```

```
--
-- Function: opWithProfileSeqPrint
--
procedure opWithProfileSeqPrint(owp_seq: in OpWithProfileSeq) is
begin
    foreach((owp: OpWithProfile), owp_sequence_pkg.scan, (owp_seq),
        put(convert(name(owp.op)));
        put(": ");
        foreach((the_id: psdl_id; the_tn: type_name),
                type_declaration_pkg.scan, (inputs(owp.op)),
            put(convert(the_tn.name));
            put(" ");
        )
        put("-> ");
            foreach((the_id: psdl_id; the_tn: type_name),
                type_declaration_pkg.scan, (outputs(owp.op)),
            put(convert(the_tn.name));
            put(" ");
        )
        put("   ");
        profilePut(software_base.getProfile(owp.op_profile));
        new_line;
    )
end opWithProfileSeqPrint;

--
-- Function: opWithProfileSetPrint
--
procedure opWithProfileSetPrint(owp_set: in OpWithProfileSet) is
begin
    foreach((owp: OpWithProfile), owp_set_pkg.scan, (owp_set),
        put(convert(name(owp.op)));
        put(": ");
        foreach((the_id: psdl_id; the_tn: type_name),
                type_declaration_pkg.scan, (inputs(owp.op)),
            put(convert(the_tn.name));
            put(" ");
        )
        put("-> ");
            foreach((the_id: psdl_id; the_tn: type_name),
                type_declaration_pkg.scan, (outputs(owp.op)),
            put(convert(the_tn.name));
            put(" ");
        )
        new_line;
        profilePut(software_base.getProfile(owp.op_profile));
        new_line;
    )
end opWithProfileSetPrint;

--
-- Function: addOpWithProfile
--
procedure addOpWithProfile(owp: in OpWithProfile;
    owp_seq: in out OpWithProfileSeq) is
begin
    owp_sequence_pkg.add(owp, owp_seq);
    owp_seq := opWithProfileSeqSort(owp_seq);
end addOpWithProfile;


--
-- Function: createNumericSignatures
--
-- Description: helper function to create numeric signatures for
--              an operator.
--
function createNumericSignatures(op: in operator;
    generics_mapping: GenericsMap; type_id: psdl_id)
    return SignatureSequence is

    package type_map_pkg is
        new generic_map_pkg(
        key => type_name,
        result => integer,
        eq_key => equal,
        eq_res => "=",
        average_size => 2);
    subtype type_map is type_map_pkg.map;

    -- if a type from the same sort group is already in the map
    -- then return the number that represents that sort group
    -- otherwise return 0, indicating this a type from a new
    -- sort group
    function getSortGroupNum(the_type: type_name;
            the_type_map: type_map) return integer is
        return_val: integer;
    begin
        return_val := 0;
        foreach((the_tn: type_name; the_num: integer),
                type_map_pkg.scan, (the_type_map),
            if same_sort_group(the_type, the_tn) then
                return_val := the_num;
                -- TODO: should be exit loop here but don't know how to
            end if;
        )
        return return_val;
    end getSortGroupNum;

    the_inputs: type_declaration := inputs(op);
    the_outputs: type_declaration := outputs(op);
    the_type_map: type_map;
    i, t: natural;
    sort_group_num: integer;
    gen_set: psdl_id_set_subtype_pkg.psdl_id_set;
    temp_signature: Signature;
    temp_tn: type_name;
    return_val: SignatureSequence;
    type_occurrence_count: natural;
    bool_count, char_count, string_count, int_count, float_count: natural;

    procedure update_additional_counts(the_tn: type_name) is
    begin
        if eq(temp_tn.name, type_id) then
            type_occurrence_count := type_occurrence_count + 1;
        elsif same_sort_group(the_tn, boolean_type) then
            bool_count := bool_count + 1;
        elsif same_sort_group(the_tn, character_type) then
            char_count := char_count + 1;
        elsif same_sort_group(the_tn, string_type) then
            string_count := string_count + 1;
        elsif same_sort_group(the_tn, integer_type) then
            int_count := int_count + 1;
        elsif same_sort_group(the_tn, float_type) then
            float_count := float_count + 1;
        end if;
    end;

begin

    type_map_pkg.create(0, the_type_map);

    -- for each output
    foreach((o_id: psdl_id; o_tn: type_name),
            type_declaration_pkg.scan, (the_outputs),
        type_map_pkg.recycle(the_type_map);
        t := 0;
        i := 0;
        type_occurrence_count := 0;
        bool_count := 0;
        char_count := 0;
        string_count := 0;
        int_count := 0;
        float_count := 0;

        -- for each input
        foreach((i_id: psdl_id; i_tn: type_name),
                type_declaration_pkg.scan, (the_inputs),

            -- check if type is a generic type or a regular type
            if generics_map_pkg.member(i_tn.name, generics_mapping) then
            temp_tn := create(
                generics_map_pkg.fetch(generics_mapping, i_tn.name),
                psdl_id_sequence_pkg.empty,
                type_declaration_pkg.create(null_type));
        else
            -- could probably use i_tn as is rather than create
            -- a copy but we're being safe in case i_tn has some
            -- residue in its formals and gen_pars
            temp_tn := create(i_tn.name,
                psdl_id_sequence_pkg.empty,
                type_declaration_pkg.create(null_type));
        end if;

        update_additional_counts(temp_tn);

        -- if the type isn't in the map yet then put it in
        if not type_map_pkg.member(temp_tn, the_type_map) then
            sort_group_num := getSortGroupNum(temp_tn, the_type_map);
            if sort_group_num = 0 then
                t := t + 1;
                type_map_pkg.bind(temp_tn, t, the_type_map);
            end if;
        end if;

        -- add the input's sort group number
        i := i + 1;
        temp_signature(i) := getSortGroupNum(temp_tn, the_type_map);
    )

    -- handle the output

    -- check if type is a generic type or a regular type
    if generics_map_pkg.member(o_tn.name, generics_mapping) then
        temp_tn := create(
            generics_map_pkg.fetch(generics_mapping, o_tn.name),
            psdl_id_sequence_pkg.empty,
            type_declaration_pkg.create(null_type));
    else
        -- could probably use o_tn as is rather than create
        -- a copy but we're being safe in case o_tn has some
        -- residue in its formals and gen_pars
        temp_tn := create(o_tn.name,
            psdl_id_sequence_pkg.empty,
            type_declaration_pkg.create(null_type));
    end if;

    update_additional_counts(temp_tn);

    -- if the type isn't in the map yet then put it in
    if not type_map_pkg.member(temp_tn, the_type_map) then
        sort_group_num := getSortGroupNum(temp_tn, the_type_map);
        if sort_group_num = 0 then
            t := t + 1;
            type_map_pkg.bind(temp_tn, t, the_type_map);
        end if;
    end if;

    -- add the output's sort group number
    i := i + 1;
    temp_signature(i) := getSortGroupNum(temp_tn, the_type_map);

    -- mark end of signature
    i := i + 1;
    temp_signature(i) := 0;

    -- add the type_occurrence_count to the signature
    i := i + 1;
    temp_signature(i) := type_occurrence_count;

    -- add basic type counts in
    i := i + 1;
    temp_signature(i) := bool_count;
    i := i + 1;
    temp_signature(i) := char_count;
    i := i + 1;
    temp_signature(i) := string_count;
```

```
        i := i + 1;
        temp_signature(i) := int_count;
        i := i + 1;
        temp_signature(i) := float_count;

        i := i + 1;
        temp_signature(i) := 0;

        -- add the signature to the sequence of signatures
        signature_seq_pkg.add(temp_signature, return_val);
    )

    return return_val;
end createNumericSignatures;


--
-- Function: getOperatorProfiles
--
-- Description: helper function to collect the profiles for
--              an operator.  A ComponentProfile (sequence of
--              profiles) is used because if an operator has
--              more than one output it is treated as if there
--              is a separate operator for each output.
--
function getOperatorProfiles(op: operator;
        generics_mapping: in GenericsMap; type_id: psdl_id)
        return ComponentProfile is

    return_val: ComponentProfile;
    numeric_sigs: SignatureSequence;

begin
    -- convert the operator's signature to numeric signatures
    -- (see the comments in the specification of profile_calc)
    numeric_sigs := createNumericSignatures(op, generics_mapping, type_id);

    -- compute the profile for each signature
    foreach((sig: Signature), signature_seq_pkg.scan, (numeric_sigs),
        addProfileID(software_base.getProfileID(computeProfile(sig)),
        return_val);
    )

    return return_val;
end getOperatorProfiles;


--
-- Function: getComponentProfile
--
-- Description: this function will return the ComponentProfile
--              for a component specified in PSDL in the PSDL
--              file filename.
--
function getComponentProfile(filename: in string;
        generics_mapping: in GenericsMap) return ComponentProfile is

    the_file: file_type;
    the_prog: psdl_program;
    return_val: ComponentProfile;

begin
    -- parse the psdl file to create a psdl_program
    open(the_file, IN_FILE, filename);
    assign(the_prog, psdl_program_pkg.empty_psdl_program);
    psdl_io.get(the_file, the_prog);
    close(the_file);

    -- if the program contains more than one component
    -- then just get the first one since the program
    -- is only supposed to have one (a requirement of
    -- this implementation)
    foreach((c_id: psdl_id; c: psdl_component),
        psdl_program_map_pkg.scan, (the_prog),

        -- if the component is a single operator then just
        -- get the profile for that operator
        if component_category(c) = psdl_operator then
            addProfiles(getOperatorProfiles(c, generics_mapping, empty),
                return_val);

        -- otherwise the component is a type so get the profiles
        -- for each of its operators
        else
            foreach((id: psdl_id; o: operator),
                operation_map_pkg.scan, (operations(c)),

                addProfiles(getOperatorProfiles(o, generics_mapping,
                    psdl_id_pkg.Upper_To_Lower(c_id)), return_val);
            )
        end if;

        -- TODO: need to break out of this loop so that only the
        --       first component is processed.
    )

    return return_val;
end getComponentProfile;


--
-- Function: splitOp
--
-- Description: helper function to split an operator with more
--              than one output into a sequence of operators
--              where each operator has one of the outputs.
--              When splitting, instances of the operator's generic
--              types in the inputs and the outpus are converted to
--              their mapped types according to the generics_mapping.
--              Each split operator's profile is then calculated.
--
function splitOp(op: operator; generics_mapping: in GenericsMap;
        type_id: psdl_id)
    return OpWithProfileSeq is

    return_val: OpWithProfileSeq;
    temp_owp: OpWithProfile;
    temp_output_name: psdl_id;
    temp_output_type: type_name;
    numeric_sigs: SignatureSequence;

begin
    -- for each output
    foreach((o_id: psdl_id; o_tn: type_name),
        type_declaration_pkg.scan, (outputs(op)),

        -- make a copy of op but with only the current output
        temp_owp.op := make_atomic_operator(
            psdl_name => name(op),
            ada_name => ada_name(op),
            gen_par => generic_parameters(op),
            keywords => keywords(op),
            axioms => axioms(op),
            state => states(op));

        -- add the inputs
        foreach((i_id: psdl_id; i_tn: type_name),
                type_declaration_pkg.scan, (inputs(op)),
            if generics_map_pkg.member(i_tn.name, generics_mapping) then
                add_input(i_id, create(
                    generics_map_pkg.fetch(generics_mapping, i_tn.name),
                        psdl_id_sequence_pkg.empty,
                        type_declaration_pkg.create(null_type)),
                    temp_owp.op);
            else
                add_input(i_id, i_tn, temp_owp.op);
            end if;
        )

        -- add the output
        if generics_map_pkg.member(o_tn.name, generics_mapping) then
            add_output(o_id, create(
            generics_map_pkg.fetch(generics_mapping, o_tn.name),
                psdl_id_sequence_pkg.empty,
                type_declaration_pkg.create(null_type)),
            temp_owp.op);
        else
            add_output(o_id, o_tn, temp_owp.op);
        end if;

        -- Convert the new operator's signature to numeric signatures
        -- (see the comments in the specification of profile_calc).
        -- Note the call to createNumericSignatures can now just pass
        -- an empty GenericsMap since the generics were mapped to actual
        -- types in the above code.
        numeric_sigs :=
            createNumericSignatures(temp_owp.op,
                generics_map_pkg.create(empty), type_id);

        -- compute the new operator's profile
        temp_owp.op_profile := software_base.getProfileID(computeProfile(
                signature_seq_pkg.fetch(numeric_sigs, 1)));

        -- add the new operator-with-profile to return_val
        addOpWithProfile(temp_owp, return_val);
    )

    return return_val;
end splitOp;


--
-- Function: getOpsWithProfiles
--
-- Description: constructs a sequence of OpWithProfiles (a PSDL operator
--              and its corresponding profile) representing the operators
--              in the PSDL component specified in filename.
--
function getOpsWithProfiles(filename: in string;
        generics_mapping: in GenericsMap) return OpWithProfileSeq is

    the_file: file_type;
    the_prog: psdl_program;
    return_val, foo: OpWithProfileSeq := owp_sequence_pkg.empty;

begin
    -- parse the psdl file to create a psdl_program
    open(the_file, IN_FILE, filename);
    assign(the_prog, psdl_program_pkg.empty_psdl_program);
    psdl_io.get(the_file, the_prog);
    close(the_file);

    -- if the program contains more than one component
    -- then just get the first one since the program
    -- is only supposed to have one (a requirement of
    -- this implementation).  Generic maps need a method
    -- that allows the user to fetch a single mapping
    -- in the map.
    foreach((c_id: psdl_id; c: psdl_component),
        psdl_program_map_pkg.scan, (the_prog),

        -- if the component is a single operator then just
        -- get that operator
        if component_category(c) = psdl_operator then
            foreach((owp: OpWithProfile), owp_sequence_pkg.scan,
                    (splitOp(c, generics_mapping, empty)),
                addOpWithProfile(owp, return_val);
            )

        -- otherwise the component is a type so get
        -- each of its operators
        else
            foreach((id: psdl_id; o: operator),
                operation_map_pkg.scan, (operations(c)),

                foreach((owp: OpWithProfile), owp_sequence_pkg.scan,
                        (splitOp(o, generics_mapping,
                            psdl_id_pkg.Upper_To_Lower(c_id))),
                addOpWithProfile(owp, return_val);
            )

            -- in the above statement we
            -- temporally pass the generic parameters for the whole
            -- type, c.  Should really just pass the generic
            -- parameters for the operation, o, only. This will
            -- happen when generics get reworked.
        )
    end if;

    -- TODO: need to break out of this loop so that only the
    --       first component is processed.
  )

  return return_val;
end getOpsWithProfiles;


--
-- Function: getOpsWithProfiles
--
-- Description: constructs a set of OpWithProfiles (a PSDL operator
--              and its corresponding profile) representing the operators
--              in the PSDL component specified in filename.
--
function getOpsWithProfiles(filename: in string;
        generics_mapping: in GenericsMap) return OpWithProfileSet is

    the_file: file_type;
    the_prog: psdl_program;
    return_val: OpWithProfileSet;

begin
    -- parse the psdl file to create a psdl_program
    open(the_file, IN_FILE, filename);
    assign(the_prog, psdl_program_pkg.empty_psdl_program);
    psdl_io.get(the_file, the_prog);
    close(the_file);

    -- if the program contains more than one component
    -- then just get the first one since the program
    -- is only supposed to have one (a requirement of
    -- this implementation).  Generic maps need a method
    -- that allows the user to fetch a single mapping
    -- in the map.
    foreach((c_id: psdl_id; c: psdl_component),
        psdl_program_map_pkg.scan, (the_prog),

        -- if the component is a single operator then just
        -- get that operator
        if component_category(c) = psdl_operator then
            foreach((owp: OpWithProfile), owp_sequence_pkg.scan,
                    (splitOp(c, generics_mapping, empty)),
                owp_set_pkg.add(owp, return_val);
            )

        -- otherwise the component is a type so get
        -- each of its operators
        else
            foreach((id: psdl_id; o: operator),
                operation_map_pkg.scan, (operations(c)),

                foreach((owp: OpWithProfile), owp_sequence_pkg.scan,
                        (splitOp(o, generics_mapping,
                            psdl_id_pkg.Upper_To_Lower(c_id))),
                    owp_set_pkg.add(owp, return_val);
                )

                -- in the above statement we
                -- temporally pass the generic parameters for the whole
                -- type, c.  Should really just pass the generic
                -- parameters for the operation, o, only. This will
                -- happen when generics get reworked.
            )
        end if;
        -- TODO: need to break out of this loop so that only the
        --       first component is processed.
    )

    return return_val;
end getOpsWithProfiles;

--
-- Procedure: psdl_idPut
--
procedure psdl_idPut(the_id: in psdl_id) is
begin
    put(convert(the_id));
end psdl_idPut;

--
-- Function: getGenericsMap
--
-- Description: generates all the possible mappings of generic types
--              to actual types for all the generic parameters in
--              the component specified in the PSDL file, filename.
--              See description of GenericsMap in psdl_profile.ads.
--             This is done by collecting all the types used in the
--             operatations of the component (note we are only processing
--             type components, not operator components) into a set
--             and then performing the cross-product of this set with
--             the set of generic parameters.
--
function getGenericsMaps(filename: in string) return GenericsMapSet is

    the_file: file_type;
    the_prog: psdl_program;
    return_val: GenericsMapSet;
    gen_set: psdl_id_set;
    type_set: psdl_id_set;
    temp_map: GenericsMap;

    procedure cross_product(g_set, t_set: psdl_id_set; gens_map: GenericsMap) is
        temp_set: psdl_id_set;
        g: psdl_id;
        local_map: GenericsMap;
    begin
        generics_map_pkg.assign(local_map, gens_map);
        if psdl_id_set_pkg.size(g_set) > 0 then
            psdl_id_set_pkg.assign(temp_set, g_set);
            g := psdl_id_set_pkg.choose(g_set);
            foreach((the_type_id: psdl_id), psdl_id_set_pkg.scan, (t_set),
                generics_map_pkg.bind(g, the_type_id, local_map);
                psdl_id_set_pkg.remove(g, temp_set);
                cross_product(temp_set, t_set, local_map);
                generics_map_pkg.assign(local_map, gens_map);
            )
            generics_map_pkg.recycle(temp_map);
        else
            generics_map_set_pkg.add(local_map, return_val);
        end if;
    end cross_product;

begin
    return_val := generics_map_set_pkg.empty;

    -- parse the psdl file to create a psdl_program
    open(the_file, IN_FILE, filename);
    assign(the_prog, psdl_program_pkg.empty_psdl_program);
    psdl_io.get(the_file, the_prog);
    close(the_file);

    -- if the program contains more than one component
    -- then just get the first one since the program
    -- is only supposed to have one (a requirement of
    -- this implementation). Generic maps need a method
    -- that allows the user to fetch a single mapping
    -- in the map.
    foreach((c_id: psdl_id; c: psdl_component), psdl_program_map_pkg.scan,
            (the_prog),
            -- collect the names of the generic parameters
            foreach((the_id: psdl_id; the_tn: type_name),
                    type_declaration_pkg.scan, (generic_parameters(c)),
                if eq(psdl_id_pkg.Upper_To_Lower(the_tn.name),
                        convert("private_type")) then
                    psdl_id_set_pkg.add(psdl_id_pkg.Upper_To_Lower(the_id),
                        gen_set);
                end if;
            )

            -- collect the types used in all the operators
            if component_category(c) = psdl_type then
                foreach((o_id: psdl_id; o: operator),
                        operation_map_pkg.scan, (operations(c)),

                    -- inputs
                    foreach((the_id: psdl_id; the_tn: type_name),
                            type_declaration_pkg.scan, (inputs(o)),
                        psdl_id_set_pkg.add(
                            psdl_id_pkg.Upper_To_Lower(the_tn.name), type_set);
                    )

                    -- outputs
                    foreach((the_id: psdl_id; the_tn: type_name),
                            type_declaration_pkg.scan, (outputs(o)),
                        psdl_id_set_pkg.add(
                            psdl_id_pkg.Upper_To_Lower(the_tn.name), type_set);
                    )
                )
            end if;

            -- TODO: need to break out of this loop so that only the
            --       first component is processed.
        )

        generics_map_pkg.create(empty, temp_map);
        cross_product(gen_set, type_set, temp_map);

        return return_val;
    end getGenericsMaps;

end psdl_profile;
```

run_batch.g

```
----------------------------------------------------------------------
-- Program: run_batch
--
-- Description: collects statistics for measuring the effect different
--              profile definitions have on profile filtering and
--              signature matching.
----------------------------------------------------------------------
with text_io; use text_io;

with a_strings; use a_strings;
with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;

with profile_calc; use profile_calc;
with psdl_profile; use psdl_profile;
with profile_types; use profile_types;
with component_id_types; use component_id_types;
with haase_diagram; use haase_diagram;
with candidate_types; use candidate_types;
with software_base;
with sig_match_types; use sig_match_types;
with sig_match; use sig_match;

procedure run_batch is
    the_candidates: CandidateSet;
    sn, the_branch, another_branch: SigMatchNode;
    q_ops, c_ops: OpWithProfileSeq;
    batch_file: file_type;
    input_line: string(1..256);
    line_length: natural;
    queries_dir, results_dir: a_string;
    query_filename, sm_filename, p_hist_filename, sm_hist_filename: a_string;
    temp_candidate: Candidate;

    procedure printArrayProfiles(profile_array: in ArrayProfiles;
            num_profiles: in integer) is
        the_profile: ArrayProfile;
        i: integer;
        rval: integer;
    begin
        for i in 1..num_profiles loop
            the_profile := profile_array(i);
            rval := printArrayProfile(the_profile);
            new_line;
        end loop;
    end printArrayProfiles;

begin
    put_line("Initializing Software Base...");
    software_base.initialize("sb_header.txt");
    --put_line("finished.");

    put(integer'image(software_base.numComponents));
    put(" components in ");
    put(integer'image(software_base.numOccupiedPartitions));
    put_line(" partitions.");
    new_line;

    put("Generating GML...");
    software_base.generateGML("haase_diagram.gml");
    put_line("finished.");
    new_line;

    open(batch_file, in_file, "batch.txt");
    get_line(batch_file, input_line, line_length);
    queries_dir := to_a(input_line(1..line_length)) & "/queries/";
    results_dir := to_a(input_line(1..line_length)) & "/results/";
put_line(convert(text(queries_dir)));
put_line(convert(text(results_dir)));
    while (not end_of_file(batch_file)) loop
        get_line(batch_file, input_line, line_length);
        new_line;
        put("PROCESSING ");
        put_line(input_line(1..line_length));
        new_line;
        query_filename := queries_dir & to_a(input_line(1..line_length)) & ".psdl";
        p_hist_filename := results_dir & to_a(input_line(1..line_length)) & "-p-hist.txt";
        sm_hist_filename := results_dir & to_a(input_line(1..line_length)) & "-sm-
hist.txt";
        sm_filename := results_dir & to_a(input_line(1..line_length)) & "-sm-stat.txt";
        put("Profile Filtering...");
        the_candidates := software_base.profileFilter(
            convert(text(query_filename)));
        put_line("finished.");

        put(integer'image(candidate_set_pkg.size(the_candidates)));
        put_line(" candidates.");
        candidateSetPut(the_candidates);
        new_line;
        new_line;

        generateProfileHistogram(convert(text(p_hist_filename)), the_candidates);

        the_candidates := profileSkim(1.0, the_candidates);
        put(integer'image(candidate_set_pkg.size(the_candidates)));
        put_line(" candidates have profile rank >= 1.0");
        candidateSetPut(the_candidates);
        new_line;
        new_line;

        put("Signature Matching...");
        if candidate_set_pkg.size(the_candidates) > 0 then
            temp_candidate := software_base.signatureMatch(
                convert(text(query_filename)),
                candidate_set_pkg.choose(the_candidates));
            generateSigMatchHistogram(convert(text(sm_hist_filename)),
                temp_candidate);
            sigMatchStatsPut(convert(text(sm_filename)));
        end if;
    end loop;
    close(batch_file);
end run_batch;
```

sig_match.ads

```
-------------------------------------------------------------------------
-- Package Spec: sig_match
-------------------------------------------------------------------------
with psdl_profile; use psdl_profile;
with sig_match_types; use sig_match_types;

package sig_match is

 procedure match_ops(query, candidate: in OpWithProfileSeq;
    root_sn: in out SigMatchNode);

 procedure sigMatchStatsReset;
 procedure sigMatchStatsPut(filename: string);

end sig_match;
```

sig_match.g

```
-----------------------------------------------------------------------
-- Package Body: sig_match
-----------------------------------------------------------------------
with text_io; use text_io;

with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with psdl_component_pkg; use psdl_component_pkg;

with profile_types; use profile_types;
with psdl_profile; use psdl_profile;
with sig_match_types; use sig_match_types;

package body sig_match is

 failed_outputs: natural := 0;
 passed_outputs: natural := 0;
 failed_basics: natural := 0;
 passed_basics: natural := 0;
 duplicates: natural := 0;
 total_inputs: natural := 0;
 failed_inputs: natural := 0;

 --
 -- Function: get_basics
 --
 -- Description: removes any user-defined types from the inputs argument,
 --              thereby returning a type_declaration with predefined
 --              types only.
 --
 function get_basics(inputs: in type_declaration) return type_declaration is
    return_val: type_declaration;
 begin
    type_declaration_pkg.assign(return_val, inputs);
    foreach((the_id: psdl_id; the_tn: type_name), type_declaration_pkg.scan,
            (inputs),
        if not is_predefined(the_tn) then
            type_declaration_pkg.remove(the_id, return_val);
        end if;
    )
    return return_val;
 end get_basics;


 --
 -- Function: get_user_defined
 --
 -- Description: removes any predefined types from the inputs argument,
 --              thereby returning a type_declaration with user-defined
 --              types only.
 --
 function get_user_defined(inputs: in type_declaration)
       return type_declaration is
    return_val: type_declaration;
 begin
    type_declaration_pkg.assign(return_val, inputs);
    foreach((the_id: psdl_id; the_tn: type_name), type_declaration_pkg.scan,
            (inputs),
        if is_predefined(the_tn) then
            type_declaration_pkg.remove(the_id, return_val);
        end if;
    )
    return return_val;
 end get_user_defined;


 --
 -- Function: match_basics
 --
 -- Description: determines if the query's basic input types can match the
 --              candidate's basic input types given the following rule:
 --              Basic types: either they must match exactly or the
 --              query's input type must be a subtype of the component's
 --              input type.
```

```
--
function match_basics(q_basics, c_basics: in type_declaration)
      return boolean is
   the_q_basics: type_declaration;
   the_c_basics: type_declaration;
   new_q_basics: type_declaration;
   new_c_basics: type_declaration;
   found_match, found_c2, return_val: boolean;
begin
   type_declaration_pkg.assign(new_q_basics, q_basics);
   type_declaration_pkg.assign(new_c_basics, c_basics);

   --
   -- cannot match if query has different number of basics then
   -- the candidate
   --
   if type_declaration_pkg.size(q_basics) /=
           type_declaration_pkg.size(c_basics) then
       return false;
   end if;

   --
   -- filter out the basics that match exactly
   --
   type_declaration_pkg.assign(the_c_basics, new_c_basics);
   foreach((q_id: psdl_id; q_tn: type_name), type_declaration_pkg.scan,
           (q_basics),
       found_match := false;
       foreach((c_id: psdl_id; c_tn: type_name), type_declaration_pkg.scan,
               (new_c_basics),
           if not found_match then
               if equal(q_tn, c_tn) then
                   type_declaration_pkg.remove(q_id, new_q_basics);
                   type_declaration_pkg.remove(c_id, the_c_basics);
                   found_match := true;
               end if;
           end if;
           -- TODO: would rather break out of the inner for loop when a
           --       match is found rather than do this found_match stuff.
       )
       type_declaration_pkg.assign(new_c_basics, the_c_basics);
   )

   --
   -- Filter out the remaining basics that can match to supertypes.
   -- This is done by temporally mapping each query input type to a
   -- supertype in the candidate that is closest in the partial ordering
   -- of basic types.
   --
   type_declaration_pkg.assign(the_q_basics, new_q_basics);
   foreach((q_id: psdl_id; q_tn: type_name), type_declaration_pkg.scan,
           (the_q_basics),
       found_match := false;
       type_declaration_pkg.assign(the_c_basics, new_c_basics);
       foreach((c_id: psdl_id; c_tn: type_name), type_declaration_pkg.scan,
               (the_c_basics),
           if not found_match then
               if subtype_of(q_tn, c_tn) then
                   found_c2 := false;
                   foreach((c2_id: psdl_id; c2_tn: type_name),
                           type_declaration_pkg.scan, (the_c_basics),
                       if not found_c2 then
                           if not equal(c_tn, c2_tn) then
                               if subtype_of(q_tn, c2_tn) and
                                          subtype_of(c2_tn, c_tn) then
                                   found_c2 := true;
                               end if;
                           end if;
                       end if;
                   )
                   if not found_c2 then
                       type_declaration_pkg.remove(q_id, new_q_basics);
                       type_declaration_pkg.remove(c_id, new_c_basics);
                       found_match := true;
                   end if;
               end if;
```

```
            end if;
        )
    )

    --
    -- if there are any basics left over than match is not possible since
    -- basics cannot be matched to non-basics
    --
    return_val := type_declaration_pkg.size(new_q_basics) = 0;

    --
    -- recycle local variables
    --
    type_declaration_pkg.recycle(new_q_basics);
    type_declaration_pkg.recycle(new_c_basics);
    type_declaration_pkg.recycle(the_q_basics);
    type_declaration_pkg.recycle(the_c_basics);

    return return_val;
end match_basics;


--
-- Procedure: match_outputs
--
-- Description: This function serves two purposes: 1. to determine if
--              the outputs of the matched operations can match, and
--              2. if they can match, add the type mappings to sn.V.TM.
--
procedure match_outputs(sn: in out SigMatchNode; success: out boolean) is
   q_output_type, c_output_type: type_name;
begin
   success := true;
   foreach((q_op: operator; c_op: operator), op_map_pkg.scan, (sn.V.OM),
      if success then
         -- get q_op's one-and-only output type
         q_output_type := type_declaration_pkg.res_set_pkg.choose(
               type_declaration_pkg.map_range(outputs(q_op)));
         -- get c_op's one-and-only output type
         c_output_type := type_declaration_pkg.res_set_pkg.choose(
               type_declaration_pkg.map_range(outputs(c_op)));

         if is_predefined(q_output_type) or
               is_predefined(c_output_type) then
            if not subtype_of(c_output_type, q_output_type) then
               success := false;
            end if;
         elsif type_map_pkg.member(q_output_type, sn.V.TM) then
            if not equal(c_output_type,
                  type_map_pkg.fetch(sn.V.TM, q_output_type)) then
               success := false;
            end if;
         else
            type_map_pkg.bind(q_output_type, c_output_type, sn.V.TM);
         end if;
      end if;
   )
end match_outputs;


--
-- Procedure: match_inputs
--
-- Description:
--
procedure match_inputs(root_sn: in out SigMatchNode; success: out boolean) is

   procedure match(q_inputs, c_inputs: in type_declaration;
         root_sn: in out SigMatchNode; success: out boolean) is
      new_q_inputs, new_c_inputs: type_declaration;
      temp_q_inputs, temp_c_inputs: type_declaration;
      ci: type_name;
      temp_sn: SigMatchNodePtr;
      temp_id: psdl_id;
      found_temp_id: boolean;
      got_first_qi: boolean;
    return_val: SigMatchNode;
begin
    return_val := createSigMatchNode;
    sigMatchNodeAssign(return_val, root_sn);

    type_declaration_pkg.assign(new_q_inputs, q_inputs);
    type_declaration_pkg.assign(new_c_inputs, c_inputs);
    success := true;
    foreach((q_id: psdl_id; qi: type_name),
            type_declaration_pkg.scan, (q_inputs),
        if success then
            if type_map_pkg.member(qi, root_sn.V.TM) then
                ci := type_map_pkg.fetch(root_sn.V.TM, qi);
                -- if the current query input type is already mapped
                -- then make sure it is mapped to an existing type in
                -- the candidate's inputs.  Note to test this we must
                -- look at the type_declaration's range (the types)
                -- not its domain (the psdl_ids).
                if not type_declaration_pkg.res_set_pkg.member(ci,
                        type_declaration_pkg.map_range(c_inputs)) then
                    success := false;
                else
                    -- remove qi from new_q_inputs
                    type_declaration_pkg.remove(q_id, new_q_inputs);
                    -- remove ci from new_c_inputs
                    found_temp_id := false;
                    if not found_temp_id then
                        foreach((c_id: psdl_id; c_tn: type_name),
                                type_declaration_pkg.scan, (new_c_inputs),
                            if equal(ci, c_tn) then
                                temp_id := c_id;
                                found_temp_id := true;
                                -- TODO: would rather break out of for loop.
                            end if;
                        )
                    end if;
                    if found_temp_id then
                        type_declaration_pkg.remove(temp_id, new_c_inputs);
                    else
                        -- if this else block gets called
                        -- there is something wrong
                        put_line("there is something wrong");
                        success := false;
                    end if;
                end if;
            end if;
        end if;
    )
    if success then
        -- got_first_qi is a cheesy way of only getting the first
        -- element out of the map.  Maps need a way of fetching by
        -- i'th element.
        got_first_qi := false;
        foreach((q_id: psdl_id; qi: type_name),
                type_declaration_pkg.scan, (q_inputs),
            if not got_first_qi then
                got_first_qi := true;
                foreach((c_id: psdl_id; c_tn: type_name),
                        type_declaration_pkg.scan, (c_inputs),
                    temp_sn := new SigMatchNode'(createSigMatchNode);
                    sigMatchNodeAssign(temp_sn.all, root_sn);
                    temp_sn.expanded_for_inputs := false;
                    type_map_pkg.bind(qi, c_tn, temp_sn.V.TM);
                    type_declaration_pkg.assign(temp_q_inputs,
                        new_q_inputs);
                    type_declaration_pkg.assign(temp_c_inputs,
                        new_c_inputs);
                    type_declaration_pkg.remove(q_id, temp_q_inputs);
                    type_declaration_pkg.remove(c_id, temp_c_inputs);
                    match(temp_q_inputs, temp_c_inputs, temp_sn.all,
                        success);
                    if success then
                        addBranch(temp_sn, return_val);
                    end if;
                )
            end if;
            )
        end if;
        sigMatchNodeAssign(root_sn, return_val);
    end match;

    q_inputs, c_inputs: type_declaration;

begin
    success := true;
    foreach((q_op: operator; c_op: operator), op_map_pkg.scan, (root_sn.V.OM),
        if success then
            --
            -- Remove the input types that have already been mapped.
            --
            type_declaration_pkg.assign(q_inputs, inputs(q_op));
            type_declaration_pkg.assign(c_inputs, inputs(c_op));

            -- query
            foreach((the_id: psdl_id; the_tn: type_name),
                    type_declaration_pkg.scan, (inputs(q_op)),
                if type_map_pkg.key_set_pkg.member(the_tn,
                        type_map_pkg.map_domain(root_sn.V.TM)) then
                    -- If the type was mapped make sure it was mapped to
                    -- a type in the candidate operator.  This is necessary
                    -- because inputs are mapped for one operator at a time.
                    if type_declaration_pkg.res_set_pkg.member(
                            type_map_pkg.fetch(root_sn.V.TM, the_tn),
                            type_declaration_pkg.map_range(c_inputs)) then
                        type_declaration_pkg.remove(the_id, q_inputs);
                    else
                        success := false;
                    end if;
                end if;
            )

            -- candidate
            foreach((the_id: psdl_id; the_tn: type_name),
                    type_declaration_pkg.scan, (inputs(c_op)),
                if type_map_pkg.res_set_pkg.member(the_tn,
                        type_map_pkg.map_range(root_sn.V.TM)) then
                    type_declaration_pkg.remove(the_id, c_inputs);
                end if;
            )

            --
            -- if the number of remaining inputs types for the query and
            -- the candidate are not equal then the operations cannot match
            --
            if success then
                if type_declaration_pkg.size(q_inputs) /=
                        type_declaration_pkg.size(c_inputs) then
                    success := false;
                else
                    -- if the node has already been expanded for inputs then
                    -- all of its operators' inputs must already be mapped
                    -- otherwise the node fails.
                    if root_sn.expanded_for_inputs then
                        success := type_declaration_pkg.size(q_inputs) = 0;
                    else
                        match(get_user_defined(q_inputs),
                            get_user_defined(c_inputs),       root_sn, success);
                    end if;
                end if;
            end if;
        end if;
    )
end match_inputs;


--
-- Function: verify_subtypes
--
-- Description:
--
function verify_subtypes(root_sn: in SigMatchNode) return boolean is
begin
```

```
   -- TODO
   return true;
end verify_subtypes;


--
-- Procedure: match_ops
--
-- Description: this is the main procedure for signature matching.
--              Given the operations and their profiles for a query and a
--              candidate, this method will return a SigMatchNode whose
--              branches contain valid operation and type mappings.
--
procedure match_ops(query, candidate: in OpWithProfileSeq;
       root_sn: in out SigMatchNode) is
   return_val: SigMatchNode;
   temp_sn: SigMatchNodePtr;
   success, pruned: boolean;
   temp_query, temp_candidate: OpWithProfileSeq;
   temp_char: character;
begin
   return_val := createSigMatchNode;
   sigMatchNodeAssign(return_val, root_sn);

   owp_sequence_pkg.assign(temp_query, query);
   owp_sequence_pkg.assign(temp_candidate, candidate);
   foreach((q_owp: OpWithProfile), owp_sequence_pkg.scan, (query),
       foreach((c_owp: OpWithProfile), owp_sequence_pkg.scan, (candidate),
           if q_owp.op_profile = c_owp.op_profile then
               temp_sn := new SigMatchNode'(createSigMatchNode);
               sigMatchNodeAssign(temp_sn.all, root_sn);
               op_map_pkg.bind(q_owp.op, c_owp.op, temp_sn.V.OM);
               if not validPairingExists(temp_sn.V.OM, return_val) then
                   match_outputs(temp_sn.all, success);
                   if success then
                       passed_outputs := passed_outputs + 1;
                       if match_basics(get_basics(inputs(q_owp.op)),
                               get_basics(inputs(c_owp.op))) then
                           opWithProfileSeqRemove(q_owp, temp_query);
                           opWithProfileSeqRemove(c_owp, temp_candidate);
                           match_ops(temp_query, temp_candidate, temp_sn.all);
                           addBranch(temp_sn, return_val);
                           passed_basics := passed_basics + 1;
                       else
                           failed_basics := failed_basics + 1;
                       end if;
                   else
                       failed_outputs := failed_outputs + 1;
                   end if;
               else
                   duplicates := duplicates + 1;
               end if;
           end if;
       )
   )

   --
   -- prune leaf nodes until all leaves are valid solutions
   --
   pruned := true;
   while pruned loop
       pruned := false;
       sigMatchNodeAssign(root_sn, return_val);
       foreach((leaf_snp: SigMatchNodePtr), sig_match_node_ptr_seq_pkg.scan,
               (getLeafNodePtrs(root_sn)),
           if leaf_snp.validation = UNKNOWN then
               match_inputs(leaf_snp.all, success);
               total_inputs := total_inputs + 1;
               if not success then
                   leaf_snp.validation := INVALID;
               elsif not verify_subtypes(leaf_snp.all) then
                   leaf_snp.validation := INVALID;
               else
                   if sig_match_node_ptr_seq_pkg.length(
                           leaf_snp.branches) = 0 then
                       leaf_snp.validation := VALID;
```

```
                    else
                        leaf_snp.expanded_for_inputs := true;
                    end if;
                end if;
                if leaf_snp.validation = INVALID then
                    -- removeBranch(leaf_snp, return_val);
                    removeAllMatchingBranches(leaf_snp, return_val);
                    failed_inputs := failed_inputs + 1;
                    pruned := true;
                end if;
            end if;
        )
    end loop;

    --
    -- recycle local variables
    --
    owp_sequence_pkg.recycle(temp_query);
    owp_sequence_pkg.recycle(temp_candidate);

    sigMatchNodeAssign(root_sn, return_val);
 end match_ops;

 procedure sigMatchStatsReset is
 begin
    failed_outputs := 0;
    passed_outputs := 0;
    failed_basics := 0;
    passed_basics := 0;
    duplicates := 0;
    total_inputs := 0;
    failed_inputs := 0;
 end sigMatchStatsReset;

 procedure sigMatchStatsPut(filename: string) is
    the_file: file_type;
 begin
    create(the_file, out_file, filename);
    put(the_file, "Duplicates: ");
    put_line(the_file, integer'image(duplicates));
    put(the_file, "Passed Output Matching: ");
    put_line(the_file, integer'image(passed_outputs));
    put(the_file, "Failed Output Matching: ");
    put_line(the_file, integer'image(failed_outputs));
    put(the_file, "Passed Predefined Type Matching: ");
    put_line(the_file, integer'image(passed_basics));
    put(the_file, "Failed Predefined Type Matching: ");
    put_line(the_file, integer'image(failed_basics));
    put(the_file, "Total Inputs: ");
    put_line(the_file, integer'image(total_inputs));
    put(the_file, "Failed Inputs: ");
    put_line(the_file, integer'image(failed_inputs));
    close(the_file);
 end sigMatchStatsPut;

end sig_match;
```

sig_match_types.ads

```
--------------------------------------------------------------------------
-- Package Spec: sig_match_types
--------------------------------------------------------------------------
with text_io; use text_io;

with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with psdl_component_pkg; use psdl_component_pkg;

with generic_map_pkg;
with generic_sequence_pkg;
with generic_set_pkg;
with ordered_set_pkg;

package sig_match_types is

 --
 -- Types
 --

 --
 -- TypeMap
 --
 package type_map_pkg is new generic_map_pkg(
    key => type_name,
    result => type_name,
    eq_key => equal,
    eq_res => equal,
    average_size => 4);
 subtype TypeMap is type_map_pkg.map;

 procedure typeNamePut(the_tn: type_name);

 procedure typeMapPut is new type_map_pkg.generic_put(
    key_put => typeNamePut, res_put => typeNamePut);

 procedure typeMapFilePut is new type_map_pkg.generic_file_put(
    key_put => typeNamePut, res_put => typeNamePut);

 --
 -- OpMap
 --
 package op_map_pkg is new generic_map_pkg(
    key => operator,
    result => operator,
    eq_key => eq,
    eq_res => eq,
    average_size => 4);
 subtype OpMap is op_map_pkg.map;

 procedure opPut(the_op: operator);

 procedure opMapPut is new op_map_pkg.generic_put(
    key_put => opPut, res_put => opPut);

 procedure opMapFilePut is new op_map_pkg.generic_file_put(
    key_put => opPut, res_put => opPut);

 --
 -- SignatureMap
 --
 type SignatureMap is record
    TM: TypeMap;
    OM: OpMap;
 end record;

 function createSignatureMap return SignatureMap;

 procedure addTypeMapping(tn1: in type_name; tn2: in type_name;
    sm: in out SignatureMap);

 procedure addOpMapping(op1: in operator; op2: in operator;
    sm: in out SignatureMap);
```

```
function signatureMapEqual(sm1: in SignatureMap; sm2: in SignatureMap)
   return boolean;

procedure signatureMapPut(sm: in SignatureMap);


--
-- SignatureMapSet
--
package sig_map_set_pkg is new generic_set_pkg(
   t => SignatureMap, eq => signatureMapEqual);
subtype SignatureMapSet is sig_map_set_pkg.set;

procedure signatureMapSetPut is
   new sig_map_set_pkg.generic_put(put => signatureMapPut);


--
-- SigMatchNodePtr
--
type SigMatchNode;
type SigMatchNodePtr is access SigMatchNode;

function sigMatchNodePtrEqual(smnp1: in SigMatchNodePtr;
   smnp2: in SigMatchNodePtr) return boolean;

function sigMatchNodePtrLessThan(smnp1: in SigMatchNodePtr;
   smnp2: in SigMatchNodePtr) return boolean;

procedure sigMatchNodePtrPut(smnp: in SigMatchNodePtr);

--
-- SigMatchNodePtrSeq
--
package sig_match_node_ptr_seq_pkg is new generic_sequence_pkg(
   t => SigMatchNodePtr, average_size => 4);
subtype SigMatchNodePtrSeq is sig_match_node_ptr_seq_pkg.sequence;

function sigMatchNodePtrSeqEqual is
   new sig_match_node_ptr_seq_pkg.generic_equal(eq => sigMatchNodePtrEqual);

function sigMatchNodePtrSeqMember is
   new sig_match_node_ptr_seq_pkg.generic_member(eq => sigMatchNodePtrEqual);

procedure sigMatchNodePtrSeqRemove is
   new sig_match_node_ptr_seq_pkg.generic_remove(eq => sigMatchNodePtrEqual);

procedure sigMatchNodePtrSeqPut is
   new sig_match_node_ptr_seq_pkg.generic_put(put => sigMatchNodePtrPut);

--
-- SigMatchNodePtrSet
--
package sig_match_node_ptr_set_pkg is new ordered_set_pkg(
   t => SigMatchNodePtr, eq => sigMatchNodePtrEqual,
   "<" => sigMatchNodePtrLessThan);
subtype SigMatchNodePtrSet is sig_match_node_ptr_set_pkg.set;

procedure sigMatchNodePtrSetPut is
   new sig_match_node_ptr_set_pkg.generic_put(put => sigMatchNodePtrPut);

procedure sigMatchNodePtrSetPrint(the_set: sigMatchNodePtrSet);

--
-- SigMatchNode
--
type ValidationType is (UNKNOWN, VALID, INVALID);
type SigMatchNode is record
   id: natural;
   signature_rank: float;
   semantic_rank: float;
   V: SignatureMap;
   validation: ValidationType;
   expanded_for_inputs: boolean;
   branches: SigMatchNodePtrSeq;
end record;
```

```
 function createSigMatchNode return SigMatchNode;

 procedure addBranch(the_branch: in SigMatchNodePtr;
    the_node: in out SigMatchNode);

 procedure removeBranch(the_branch: in SigMatchNodePtr;
    the_node: in out SigMatchNode);

 procedure removeAllMatchingBranches(the_branch: in SigMatchNodePtr;
    the_node: in out SigMatchNode);

 function sigMatchNodeEqual(smn1: in SigMatchNode; smn2: in SigMatchNode)
    return boolean;

 function sigMatchNodeLessThan(smn1: in SigMatchNode; smn2: in SigMatchNode)
    return boolean;

 procedure sigMatchNodeAssign(smn1: in out SigMatchNode;
    smn2: in SigMatchNode);

 procedure sigMatchNodePut(the_node: in SigMatchNode);

 procedure sigMatchNodePrint(the_node: SigMatchNode);

 procedure generateGML(the_node: in SigMatchNode; filename: in string);

 function getLeafNodePtrs(the_node: in SigMatchNode) return SigMatchNodePtrSeq;

 function getLeafNodePtrs(the_node: in SigMatchNode) return SigMatchNodePtrSet;

 function getValidLeafNodePtrs(the_node: in SigMatchNode)
    return SigMatchNodePtrSet;

 function validPairingExists(pairing: in OpMap; the_node: in SigMatchNode)
    return boolean;

end sig_match_types;
```

sig_match_types.g

```
-----------------------------------------------------------------------
-- Package Body: sig_match_types
-----------------------------------------------------------------------
with text_io; use text_io;
with ada.float_text_io;

with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with psdl_component_pkg; use psdl_component_pkg;

with candidate_types;

package body sig_match_types is

 --
 -- Procedure: typeNamePut
 --
 -- Description: outputs the type_name's name
 --
 procedure typeNamePut(the_tn: type_name) is
 begin
    if not equal(the_tn, null_type) then
       put(convert(the_tn.name));
    end if;
 end typeNamePut;

 --
 -- Procedure: opPut
 --
 -- Description: outputs the operator's name
 --
 procedure opPut(the_op: operator) is
 begin
    if the_op /= null_component then
       put(convert(name(the_op)));
    end if;
 end opPut;

 --
 -- Function: createSignatureMap
 --
 -- Description: create and initialize a SignatureMap for use.
 --
 function createSignatureMap return SignatureMap is
    return_val: SignatureMap;
 begin
    return_val.TM := type_map_pkg.create(null_type);
    return_val.OM := op_map_pkg.create(null_component);
    return return_val;
 end createSignatureMap;

 --
 -- Procedure: addTypeMapping
 --
 -- Description: binds two types together and adds them to the
 --              SignatureMap's TypeMap.
 --
 procedure addTypeMapping(tn1: in type_name; tn2: in type_name;
       sm: in out SignatureMap) is
 begin
    type_map_pkg.bind(tn1, tn2, sm.TM);
 end addTypeMapping;

 --
 -- Procedure: addOpMapping
 --
 -- Description: binds two operators together and adds them to the
 --              SignatureMap's OpMap.
 --
 procedure addOpMapping(op1: in operator; op2: in operator;
       sm: in out SignatureMap) is
 begin
    op_map_pkg.bind(op1, op2, sm.OM);
 end addOpMapping;
```

```
--
-- Function: signatureMapEqual
--
function signatureMapEqual(sm1: in SignatureMap; sm2: in SignatureMap)
      return boolean is
begin
   return type_map_pkg.equal(sm1.TM, sm2.TM) and
      op_map_pkg.equal(sm1.OM, sm2.OM);
end signatureMapEqual;

--
-- Function: signatureMapPut
--
procedure signatureMapPut(sm: in SignatureMap) is
begin
   put("OM: ");
   opMapPut(sm.OM);
   put(" | TM: ");
   typeMapPut(sm.TM);
end signatureMapPut;

--
-- Function: sigMatchNodePtrEqual
--
function sigMatchNodePtrEqual(smnp1: in SigMatchNodePtr;
      smnp2: in SigMatchNodePtr) return boolean is
begin
   return sigMatchNodeEqual(smnp1.all, smnp2.all);
end sigMatchNodePtrEqual;

--
-- Function: sigMatchNodePtrLessThan
--
function sigMatchNodePtrLessThan(smnp1: in SigMatchNodePtr;
      smnp2: in SigMatchNodePtr) return boolean is
begin
   return sigMatchNodeLessThan(smnp1.all, smnp2.all);
end sigMatchNodePtrLessThan;

--
-- Procedure: sigMatchNodePtrPut
--
procedure sigMatchNodePtrPut(smnp: in SigMatchNodePtr) is
begin
   sigMatchNodePut(smnp.all);
end sigMatchNodePtrPut;

--
-- Function: sigMatchNodeEqual
--
function sigMatchNodeEqual(smn1: in SigMatchNode; smn2: in SigMatchNode)
      return boolean is
begin
   if smn1.signature_rank /= smn2.signature_rank then
      return false;
   end if;

   if smn1.semantic_rank /= smn2.semantic_rank then
      return false;
   end if;

   if smn1.validation /= smn2.validation then
      return false;
   end if;

   if smn1.expanded_for_inputs /= smn2.expanded_for_inputs then
      return false;
   end if;

   if not signatureMapEqual(smn1.V, smn2.V) then
      return false;
   end if;

   return sigMatchNodePtrSeqEqual(smn1.branches, smn2.branches);
end sigMatchNodeEqual;
```

```
--
-- Function: sigMatchNodeLessThan
--
function sigMatchNodeLessThan(smn1: in SigMatchNode;
      smn2: in SigMatchNode) return boolean is
begin
   if smn1.signature_rank > smn2.signature_rank then
      return true;
   -- the following test for less-than is just being paranoid
   -- about potential float equality problems
   elsif smn1.signature_rank < smn2.signature_rank then
      return false;
   elsif smn1.semantic_rank > smn2.semantic_rank then
      return true;
   -- the following test for less-than is just being paranoid
   -- about potential float equality problems
   elsif smn1.semantic_rank < smn2.semantic_rank then
      return false;
   else
      return smn1.id < smn2.id;
   end if;
end sigMatchNodeLessThan;

--
-- Procedure: sigMatchNodeAssign
--
procedure sigMatchNodeAssign(smn1: in out SigMatchNode;
   smn2: in SigMatchNode) is
begin
   smn1.signature_rank := smn2.signature_rank;
   smn1.semantic_rank := smn2.semantic_rank;
   smn1.validation := smn2.validation;
   smn1.expanded_for_inputs := smn2.expanded_for_inputs;
   type_map_pkg.assign(smn1.V.TM, smn2.V.TM);
   op_map_pkg.assign(smn1.V.OM, smn2.V.OM);
   -- TODO: might have to do the deep copy myself here
   --       rather than call assign
   sig_match_node_ptr_seq_pkg.assign(smn1.branches, smn2.branches);
end sigMatchNodeAssign;

--
-- Procedure: sigMatchNodePut
--
procedure sigMatchNodePut(the_node: in SigMatchNode) is
begin
   put("(Signature Rank: ");
   if the_node.signature_rank = candidate_types.RANK_UNKNOWN then
      put("unknown");
   else
      ada.float_text_io.put(the_node.signature_rank, 1, 2, 0);
   end if;
   put(" | ");
   put("(Semantic Rank: ");
   if the_node.semantic_rank = candidate_types.RANK_UNKNOWN then
      put("unknown");
   else
      ada.float_text_io.put(the_node.semantic_rank, 1, 2, 0);
   end if;
   put(" | ");
   case the_node.validation is
      when UNKNOWN => put("Validation Unknown");
      when VALID => put("Valid");
      when INVALID => put("Invalid");
   end case;
   put(" | ");
   if the_node.expanded_for_inputs then
      put("Expanded");
   else
      put("Not Expanded");
   end if;
   put(" | ");
   put("Op Map: ");
   opMapPut(the_node.V.OM);
   put(" | ");
   put("Type Map: ");
   typeMapPut(the_node.V.TM);
   put(" | ");
   put("{Branches: ");
   sigMatchNodePtrSeqPut(the_node.branches);
   put("}");
   put(")");
   new_line;
end sigMatchNodePut;

--
-- Procedure: sigMatchNodePrint
--
procedure sigMatchNodePrint(the_node: SigMatchNode) is
begin
   put("Signature Rank: ");
   if the_node.signature_rank = candidate_types.RANK_UNKNOWN then
      put("unknown");
   else
      ada.float_text_io.put(the_node.signature_rank, 1, 2, 0);
   end if;
   new_line;
   put("Semantic Rank: ");
   if the_node.semantic_rank = candidate_types.RANK_UNKNOWN then
      put("unknown");
   else
      ada.float_text_io.put(the_node.semantic_rank, 1, 2, 0);
   end if;
   new_line;
   case the_node.validation is
      when UNKNOWN => put("Validation Unknown");
      when VALID => put("Valid");
      when INVALID => put("Invalid");
   end case;
   put(", ");
   if the_node.expanded_for_inputs then
      put_line("Expanded");
   else
      put_line("Not Expanded");
   end if;
   put("Op Map: ");
   opMapPut(the_node.V.OM);
   new_line;
   put("Type Map: ");
   typeMapPut(the_node.V.TM);
   new_line;
   put("Branches: ");
   sigMatchNodePtrSeqPut(the_node.branches);
   new_line;
end sigMatchNodePrint;

--
-- Function: createSigMatchNode
--
-- Description: create and initialize a SigMatchNode for use.
--              Note, a unique node id is maintained to facilitate
--              sorting when two nodes have equal signature and
--              semantic ranks.
--
unique_node_id: natural := 0;
function createSigMatchNode return SigMatchNode is
   return_val: SigMatchNode;
begin
   return_val.id := unique_node_id;
   unique_node_id := unique_node_id + 1;
   return_val.signature_rank := candidate_types.RANK_UNKNOWN;
   return_val.semantic_rank := candidate_types.RANK_UNKNOWN;
   return_val.validation := UNKNOWN;
   return_val.expanded_for_inputs := false;
   return_val.V := createSignatureMap;
   return_val.branches := sig_match_node_ptr_seq_pkg.empty;
   return return_val;
end createSigMatchNode;

--
-- Function: addBranch
--
```

```
-- Description: add a branch (a child SigMatchNode) to the SigMatchNode.
--              A branch represents a superset of the node it belongs to.
--              What this really means is the branch node contains all the
--              type and operator mappings plus of the node it belongs to
--              plus more.
--
procedure addBranch(the_branch: in SigMatchNodePtr;
      the_node: in out SigMatchNode) is
begin
   sig_match_node_ptr_seq_pkg.add(the_branch, the_node.branches);
end addBranch;

--
-- Function: removeBranch
--
-- Description:
--
procedure removeBranch(the_branch: in SigMatchNodePtr;
      the_node: in out SigMatchNode) is
begin
   sigMatchNodePtrSeqRemove(the_branch, the_node.branches);
end removeBranch;

--
-- Function: removeAllMatchingBranches
--
-- Description:
--
procedure removeAllMatchingBranches(the_branch: in SigMatchNodePtr;
      the_node: in out SigMatchNode) is
begin
   sigMatchNodePtrSeqRemove(the_branch, the_node.branches);
   foreach((branch: SigMatchNodePtr), sig_match_node_ptr_seq_pkg.scan,
          (the_node.branches),
      removeAllMatchingBranches(the_branch, branch.all);
   )
end removeAllMatchingBranches;

--
-- Procedure: generateGML
--
-- Description: generate a GML file to graphically represent the
--              SigMatchNode's relationship with its branches.
--
procedure generateGML(the_node: in SigMatchNode; filename: string) is
   id: natural := 0; -- unique ID counter
   the_id: natural; -- place holder for call to put_node_gml
   gml_file: file_type;

   function new_id return natural is
   begin
      id := id + 1;
      return id;
   end new_id;

   procedure put_node_gml(sn: in SigMatchNode; my_id: out natural) is
      child_id: natural;
   begin
      my_id := new_id;
      put(gml_file, "node [ id ");
      put(gml_file, integer'image(my_id));
      put(gml_file, " label """);
      opMapFilePut(gml_file, sn.V.OM);
      put_line(gml_file, "\");
      typeMapFilePut(gml_file, sn.V.TM);
      put_line(gml_file, "\");
      case sn.validation is
          when UNKNOWN => put(gml_file, "Validation Unknown");
          when VALID => put(gml_file, "Valid");
          when INVALID => put(gml_file, "Invalid");
      end case;
      put_line(gml_file, "\");
      if sn.expanded_for_inputs then
          put(gml_file, "Expanded");
      else
          put(gml_file, "Not Expanded");
```

```
        end if;
        put_line(gml_file, """ ]");

        -- recursively call put_node_gml for each of its branches
        foreach((branch: SigMatchNodePtr), sig_match_node_ptr_seq_pkg.scan,
                (sn.branches),
            put_node_gml(branch.all, child_id);

            -- make the edge to the branch
            put(gml_file, "edge [ id ");
            put(gml_file, integer'image(new_id));
            put(gml_file, " source ");
            put(gml_file, integer'image(my_id));
            put(gml_file, " target ");
            put(gml_file, integer'image(child_id));
            put_line(gml_file, " ]");
        )
    end put_node_gml;

begin
    create(gml_file, out_file, filename);
    put(gml_file, "graph [ id ");
    put(gml_file, integer'image(new_id));
    put_line(gml_file, " directed 1");
    put_node_gml(the_node, the_id);
    put_line(gml_file, "]");
    close(gml_file);
end generateGML;

--
-- Function: getLeafNodePtrs
--
-- Description: collect the leaf nodes of the_node into a sequence.
--
function getLeafNodePtrs(the_node: in SigMatchNode)
        return SigMatchNodePtrSeq is
    return_val: SigMatchNodePtrSeq;

    procedure processNode(smnp: in SigMatchNodePtr) is
    begin
        if sig_match_node_ptr_seq_pkg.length(smnp.branches) = 0 then
            sig_match_node_ptr_seq_pkg.add(smnp, return_val);
            return;
        end if;
        foreach((branch: SigMatchNodePtr), sig_match_node_ptr_seq_pkg.scan,
                (smnp.branches),
            processNode(branch);
        )
    end processNode;

begin
    return_val := sig_match_node_ptr_seq_pkg.empty;
    foreach((branch: SigMatchNodePtr), sig_match_node_ptr_seq_pkg.scan,
            (the_node.branches),
        processNode(branch);
    )
    return return_val;
end getLeafNodePtrs;

--
-- Function: getLeafNodePtrs
--
-- Description: collect the leaf nodes of the_node into a set.
--              Note the set will keep duplicates out.
--
function getLeafNodePtrs(the_node: in SigMatchNode)
        return SigMatchNodePtrSet is
    return_val: SigMatchNodePtrSet;

    procedure processNode(smnp: in SigMatchNodePtr) is
    begin
        if sig_match_node_ptr_seq_pkg.length(smnp.branches) = 0 then
            sig_match_node_ptr_set_pkg.add(smnp, return_val);
            return;
        end if;
        foreach((branch: SigMatchNodePtr), sig_match_node_ptr_seq_pkg.scan,
            (smnp.branches),
         processNode(branch);
      )
   end processNode;

begin
   return_val := sig_match_node_ptr_set_pkg.empty;
   foreach((branch: SigMatchNodePtr), sig_match_node_ptr_seq_pkg.scan,
         (the_node.branches),
      processNode(branch);
   )
   return return_val;
end getLeafNodePtrs;

--
-- Function: getValidLeafNodePtrs
--
-- Description: collect the valid leaf nodes of the_node into a set.
--              Note the set will keep duplicates out.
--
function getValidLeafNodePtrs(the_node: in SigMatchNode)
      return SigMatchNodePtrSet is
   return_val: SigMatchNodePtrSet;

   procedure processNode(smnp: in SigMatchNodePtr) is
   begin
      if sig_match_node_ptr_seq_pkg.length(smnp.branches) = 0 then
         if smnp.validation = VALID then
            sig_match_node_ptr_set_pkg.add(smnp, return_val);
         end if;
         return;
      end if;
      foreach((branch: SigMatchNodePtr), sig_match_node_ptr_seq_pkg.scan,
            (smnp.branches),
         processNode(branch);
      )
   end processNode;

begin
   return_val := sig_match_node_ptr_set_pkg.empty;
   foreach((branch: SigMatchNodePtr), sig_match_node_ptr_seq_pkg.scan,
         (the_node.branches),
      processNode(branch);
   )
   return return_val;
end getValidLeafNodePtrs;

--
-- Function: validPairingExists
--
-- Description: gets all the valid leaf nodes and checks if the pairing
--              exists in any of them
--
function validPairingExists(pairing: in OpMap; the_node: in SigMatchNode)
      return boolean is
   return_val: boolean;
begin
   return_val := false;
   foreach((sn: SigMatchNodePtr), sig_match_node_ptr_set_pkg.scan,
         (getValidLeafNodePtrs(the_node)),
      if not return_val then
         return_val := op_map_pkg.submap(pairing, sn.V.OM);
         -- TODO: if return_val is true then should immediately return
         --       but for each doesn't let me do this
      end if;
   )
   return return_val;
end validPairingExists;

--
-- Procedure: sigMatchNodePtrSetPrint
--
procedure sigMatchNodePtrSetPrint(the_set: sigMatchNodePtrSet) is
begin
   foreach((the_node: SigMatchNodePtr), sig_match_node_ptr_set_pkg.scan,
         (the_set),
        sigMatchNodePrint(the_node.all);
        new_line;
    )
 end sigMatchNodePtrSetPrint;

end sig_match_types;
```

software_base.ads

```
-------------------------------------------------------------------------
-- Package Spec: software_base
-------------------------------------------------------------------------

with component_id_types; use component_id_types;
with haase_diagram; use haase_diagram;
with candidate_types; use candidate_types;
with profile_types; use profile_types;

package software_base is

 procedure initialize(header_filename: in string);

 function numComponents return natural;

 function numPartitions return natural;

 function numOccupiedPartitions return natural;

 procedure generateGML(gml_filename: in string);

 function profileFilter(query_filename: in string) return CandidateSet;

 function signatureMatch(query_filename: in string;
    the_candidate: in Candidate) return Candidate;

 function getProfileID(p: Profile) return ProfileID;

 function getProfile(p_id: ProfileID) return Profile;

 function getProfileIDs return profile_lookup_table_pkg.res_set;

private

 --
 -- the_component_id_map
 --
 the_component_id_map: ComponentIDMap;

 --
 -- the_haase_diagram
 --
 the_haase_diagram: HaaseDiagram;

 --
 -- the_profile_lookup_table
 --
 the_profile_lookup_table: ProfileLookupTable;

end software_base;
```

software_base.g

```
----------------------------------------------------------------------
-- Package Body: software_base
----------------------------------------------------------------------

with text_io; use text_io;
with ada.integer_text_io; use ada.integer_text_io;

with a_strings;
with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;

with component_id_types; use component_id_types;
with haase_diagram; use haase_diagram;
with candidate_types; use candidate_types;
with profile_types; use profile_types;
with psdl_profile; use psdl_profile;
with sig_match_types; use sig_match_types;
with profile_filter_pkg;
with sig_match;

package body software_base is

 --
 -- Procedure: initialize
 --
 -- Description: reads the header file to construct the_component_id_map
 --              and the_haase_diagram.
 --
 procedure initialize(header_filename: in string) is
    use a_strings;

    header_file: file_type;
    comp_id: ComponentID;
    dir_name: a_string;
    input_line: string(1..256);
    line_length: natural;
    comp_id_last : natural;
    temp_comp_profile: ComponentProfile;
    temp_haase_node: HaaseNode;
    temp_component: Component;
    the_generics_maps: GenericsMapSet;
    generics_mapping: GenericsMap;

    id: natural := 0;
    old_start: natural := 0;
    function new_id(start: natural) return natural is
    begin
        if start /= old_start then
            id := 0;
            old_start := start;
        end if;
        id := id + 1;
        return start + id;
    end new_id;

 begin
    --
    -- parse header file and construct the_component_id_map
    --
    component_id_map_pkg.create(createComponent, the_component_id_map);

    open(header_file, in_file, header_filename);
    while (not end_of_file(header_file)) loop
        get_line(header_file, input_line, line_length);
        get(input_line, comp_id, comp_id_last);

        -- trim spaces before and after directory name
        dir_name := reverse_order(trim(
            reverse_order(trim(a_strings.to_a(
            input_line(comp_id_last+1..line_length))))));

put("preparing ");
put(dir_name.s);
put("...");
```

```
        -- create a component for each generic_mapping
        the_generics_maps := getGenericsMaps(convert(text(dir_name & "/PSDL_SPEC")));
put(integer'image(generics_map_set_pkg.size(the_generics_maps)));
put(" components...");
        foreach((the_map: GenericsMap), generics_map_set_pkg.scan,
                (the_generics_maps),
            temp_component := createComponent;
            temp_component.psdl_filename := text(dir_name & "/PSDL_SPEC");
            generics_map_pkg.assign(temp_component.generics_mapping, the_map);
            component_id_map_pkg.bind(new_id(comp_id), temp_component,
                the_component_id_map);
        )
put_line("done");
    end loop;
    close(header_file);

    --
    -- Create the ProfileLookupTable
    --
    the_profile_lookup_table :=
        profile_lookup_table_pkg.create(DEFAULT_PROFILE_ID);

    --
    -- construct haase diagram
    --
    the_haase_diagram := createHaaseDiagram;

    -- for each item in the_component_id_map, get the component's
    -- profile and add it to the_haase_diagram
    foreach((the_comp_id: ComponentID; the_component: Component),
            component_id_map_pkg.scan, (the_component_id_map),

put("inserting ");
put(integer'image(the_comp_id));
put("...");
        temp_comp_profile := getComponentProfile(
            convert(the_component.psdl_filename), the_component.generics_mapping);

        --  check if haase node with temp_comp_profile as its key
        --  already exists.  If it does then add the component id
        --  to that node rather than make a new node.
        if haase_node_map_pkg.member(temp_comp_profile, the_haase_diagram) then
            temp_haase_node := haase_node_map_pkg.fetch(the_haase_diagram,
                temp_comp_profile);
        else
            temp_haase_node := createHaaseNode(temp_comp_profile);
        end if;
        addComponent(the_comp_id, temp_haase_node);
        addHaaseNode(temp_haase_node, the_haase_diagram);
put_line("done");
    )

put("Profile Lookup Table: ");
profileLookupTablePut(the_profile_lookup_table);
new_line;

put("adding base nodes...");
    addBaseNodes(the_haase_diagram);
put_line("done");
put("connecting nodes...");
    connectNodes(the_haase_diagram);
put_line("done");
 end initialize;


 --
 -- Function: numComponents
 --
 -- Description: return the number of components in the software base.
 --
 function numComponents return natural is
    return_val: natural;
 begin
    return component_id_map_pkg.size(the_component_id_map);
 end numComponents;
```

```
--
-- Function: numPartitions
--
-- Description: return the number of partitions in the software base.
--
function numPartitions return natural is
begin
   return haase_node_map_pkg.size(the_haase_diagram);
end numPartitions;


--
-- Function: numOccupiedPartitions
--
-- Description: return the number of occupied partitions in the
--              software base.
--
function numOccupiedPartitions return natural is
   return_val: natural := 0;
begin
   foreach((the_key: ComponentProfile; the_hn: HaaseNode),
           haase_node_map_pkg.scan, (the_haase_diagram),
       if component_id_set_pkg.size(the_hn.components) > 0 then
           return_val := return_val + 1;
       end if;
   )
   return return_val;
end numOccupiedPartitions;


--
-- Function: generateGML
--
procedure generateGML(gml_filename: string) is
begin
   generateGML(the_haase_diagram, gml_filename);
end generateGML;


--
-- Function: profileFilter
--
-- Description: performs profile filtering with the PSDL specified query
--              and returns an ordered set of candidates with the highest
--              profile ranking first.
--              Note the PSDL query must NOT contain generics.
--
function profileFilter(query_filename: in string) return CandidateSet is
   query_profile: ComponentProfile;
begin
   query_profile := getComponentProfile(query_filename,
       generics_map_pkg.create(empty));
   return profile_filter_pkg.findCandidates(query_profile, the_haase_diagram);
end profileFilter;


--
-- Function: signatureMatch
--
-- Description: performs signature matching between the PSDL specified
--              query and the_candidate and returns a copy of the_candidate
--              with the signature_matches field set.
--
function signatureMatch(query_filename: in string;
       the_candidate: in Candidate) return Candidate is
   q_ops, c_ops: OpWithProfileSeq;
   sn: SigMatchNode;
   temp_snp_set: SigMatchNodePtrSet;
   temp_component: Component;
   return_val: Candidate;
begin
   -- get the query's operators
   q_ops := getOpsWithProfiles(query_filename, generics_map_pkg.create(empty));
new_line;
put_line("Query: ");
```

```
opWithProfileSeqPrint(q_ops);
new_line;

    -- get the candidate's operators
    temp_component := component_id_map_pkg.fetch(the_component_id_map,
        the_candidate.component_id);
    c_ops := getOpsWithProfiles(convert(temp_component.psdl_filename),
        temp_component.generics_mapping);
put("Candidate: ");
put_line(integer'image(the_candidate.component_id));
put("Generics Mapping: ");
genericsMapPut(temp_component.generics_mapping);
new_line;
new_line;
opWithProfileSeqPrint(c_ops);
new_line;

    -- perform signature matching
    sn := createSigMatchNode;
    sig_match.sigMatchStatsReset;
    sig_match.match_ops(q_ops, c_ops, sn);

    -- calculate the signature ranks
    sig_match_node_ptr_set_pkg.assign(temp_snp_set, getLeafNodePtrs(sn));
    foreach((smnp: SigMatchNodePtr), sig_match_node_ptr_set_pkg.scan,
            (temp_snp_set),
        smnp.signature_rank := float(op_map_pkg.size(smnp.V.OM)) /
            float(owp_sequence_pkg.length(q_ops));
--
-- The following calculation for signature rank measures how well the
-- signature matching method works on its own.  The calculation above
-- is really a mixture of profile filtering AND signature matching.
--
--      smnp.signature_rank := float(op_map_pkg.size(smnp.V.OM)) /
--          (return_val.profile_rank * float(owp_sequence_pkg.length(q_ops)));
    )

    -- add each SigMatchNodePtr to make sure return_val's signature_matches
    -- field is sorted
    candidateAssign(return_val, the_candidate);
    foreach((smnp: SigMatchNodePtr), sig_match_node_ptr_set_pkg.scan,
            (temp_snp_set),
        sig_match_node_ptr_set_pkg.add(smnp, return_val.signature_matches);
    )

    return return_val;
 end signatureMatch;

 --
 -- Function: getProfileID
 --
 -- Description: if the profile doesn't exist then add it first then
 --              return its id.  A new id is obtained from the global
 --              variable unique_profile_id.
 --
 unique_profile_id: ProfileID := 0;

 function getProfileID(p: Profile) return ProfileID is
    return_val: ProfileID;
 begin
    return_val :=
        profile_lookup_table_pkg.fetch(the_profile_lookup_table, p);
    if return_val = DEFAULT_PROFILE_ID then
        return_val := unique_profile_id;
        unique_profile_id := unique_profile_id + 1;
put("binding ");
profilePut(p);
put(" to ");
put(integer'image(return_val));
put("...");
        profile_lookup_table_pkg.bind(p, return_val, the_profile_lookup_table);
    end if;
    return return_val;
 end getProfileID;

 --
```

```
-- Function: getProfile
--
function getProfile(p_id: ProfileID) return Profile is
   return_val: Profile;
begin
   return_val := 0;
   foreach((p: Profile; id: ProfileID), profile_lookup_table_pkg.scan,
           (the_profile_lookup_table),
       if id = p_id then
           return_val := p;
           -- TODO: should return here but for each doesn't let me
       end if;
   )
   return return_val;
end getProfile;

--
-- Function: getProfileIDs
--
function getProfileIDs return profile_lookup_table_pkg.res_set is
begin
   return profile_lookup_table_pkg.map_range(the_profile_lookup_table);
end getProfileIDs;

end software_base;
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center..2
   8725 John J. Kingman Road, Suite 0944
   Fort Belvoir, VA 22060-6218

2. Dudley Knox Library..2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943

3. Center for Naval Analysis 1
   4401 Ford Ave.
   Alexandria, VA 22302

4. Dr. Ted Lewis, Chairman, Code CS/LT..1
   Computer Science Dept.
   Naval Postgraduate School
   Monterey, CA 93943

5. Chief of Naval Research..1
   800 North Quincy St.
   Arlington, VA 22217

6. Dr. Luqi, Code CS/Lq..1
   Computer Science Dept.
   Naval Postgraduate School
   Monterey, CA 93943

7. Dr. Marvin Langston 1
   1225 Jefferson Davis Highway
   Crystal Gateway 2 / Suite 1500
   Arlington, VA 22202-4311

8. David Hislop..1
   U.S. Army Research Office
   PO Box 12211
   Research Triangle Park, NC 27709-2211

9. Capt. Talbot Manvel..1
   Naval Sea System Command
   2531 Jefferson Davis Hwy.
   Attn: TMS 378 Capt. Manvel
   Arlington, VA 22240-5150

10. CDR Michael McMahon 1
Naval Sea System Command
2531 Jefferson Davis Hwy.
Arlington, VA 22242-5160

11. Dr. Elizabeth Wald 1
Office of Naval Research
800 N. Quincy St.
ONR CODE 311
Arlington, VA 22217-5660

12. Dr. Ralph Wachter 1
Office of Naval Research
800 N. Quincy St.
CODE 311
Arlington, VA 22217-5660

13. Army Research Lab 1
115 O'Keefe Building
Attn: Mark Kendall
Atlanta, GA 30332-0862

14. National Science Foundation... 1
Attn: Bruce Barnes
Div. Computer & Computation Research
1800 G St. NW
Washington, DC 20550

15. National Science Foundation... 1
Attn: Bill Agresty
4201 Wilson Blvd.
Arlington, VA 22230

16. Hon. John W. Douglass 1
Assistant Secretary of the Navy
(Research, Devlopment and Aquisition)
Room E741
1000 Navy Pentagon
Washington, DC 20350-1000

17. Technical Library Branch 1
Naval Command, Control, and Ocean Surveillance Center
RDT&E Division, Code D0274
San Diego, CA 92152-5001

18. Head, Command and Control Department 1
Naval Command, Control and Ocean Surveillance Center
RDT&E Division, Code D40
San Diego, CA 92152-5001

19. Head, Simulation and Human Systems Technology Division 1
Naval Command, Control and Ocean Surveillance Center
RDT&E Division, Code D44
San Diego, CA 92152-5001